





2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

Módulo 2 – Desarrollo Front End estático

Índice

¿Qué es HTML?

HTML significa **Hyper Text Markup Language**, es el lenguaje más utilizado en la Web para desarrollar páginas web. HTML fue creado por Berners-Lee a finales de 1991, pero "HTML 2.0" fue la primera especificación HTML estándar que se publicó en 1995.
HTML 4.01 era una versión principal de HTML y se publicó a finales de 1999. Aunque la versión HTML 4.01 se usa ampliamente actualmente tenemos la versión **HTML-5**, que es una extensión de HTML 4.01 y esta versión se publicó en 2012.

¿Por qué aprender HTML?

Originalmente, HTML se desarrolló con la intención de definir la estructura de documentos como encabezados, párrafos, listas, etc., para facilitar el intercambio de información científica entre investigadores. Ahora, HTML se está utilizando ampliamente para formatear páginas web con la ayuda de diferentes etiquetas disponibles en lenguaje HTML.

HTML es una **OBLIGACIÓN** para que los estudiantes y los profesionales se conviertan en grandes ingenieros de software, especialmente cuando están trabajando en el dominio de desarrollo web.

Enumeraremos algunas de las ventajas claves de aprender HTML:

1. **Crear un sitio web**: podemos crear un sitio web o personalizar una plantilla web existente si conocemos bien HTML.
2. **Convertirnos en diseñadores web**: si deseamos comenzar una carrera como diseñador web profesional, el diseño de HTML y CSS es una habilidad imprescindible.
3. **Comprender la Web**: si deseamos optimizar nuestro sitio web para aumentar su velocidad y rendimiento, es bueno conocer HTML para obtener los mejores resultados.
4. **Aprender otros lenguajes**: una vez que comprendamos los conceptos básicos de HTML, otras tecnologías relacionadas como javascript, php o angular se volverán más fáciles de entender.

Aplicaciones de HTML

Como se mencionó anteriormente, HTML es uno de los lenguajes más utilizados en la web. Vamos a enumerar algunos ejemplos y aplicaciones:

- **Desarrollo de páginas web:** HTML se utiliza para crear páginas que se representan en la web. Casi todas las páginas de la web tienen etiquetas html para mostrar sus detalles en el navegador.

- **Navegación por Internet:** HTML proporciona etiquetas que se utilizan para navegar de una página a otra y se utiliza mucho en la navegación por Internet.
- **Interfaz de usuario responsiva:** las páginas HTML hoy en día funcionan bien en todas las plataformas, dispositivos móviles, pestañas, computadoras de escritorio o portátiles debido a la estrategia de diseño responsivo.
- **Las páginas HTML de soporte sin conexión,** una vez cargadas, pueden estar disponibles sin conexión en la máquina sin necesidad de Internet.
- **Desarrollo de videojuegos:** HTML5 tiene soporte nativo para una experiencia rica y ahora también es útil en el campo del desarrollo de videojuegos.

Introducción a HTML

¿Qué es HTML?

Por sus siglas en inglés HTML (Hypertext Markup Language) se lo define como un lenguaje de marcado estándar que nos permite crear y presentar páginas web de una manera versátil y eficiente.

El HTML describe la ESTRUCTURA y el CONTENIDO de una página web y es un lenguaje que consiste en etiquetas agrupadas o estructuradas de una manera lógica en función de lo que necesitamos como vista. Estas etiquetas le dicen al " navegador web " cómo debe mostrar el contenido.

HTML es un lenguaje y como tal tiene su propio vocabulario (palabras) y su propia gramática (reglas).

En la actualidad y desde hace varios años, el HTML está definido por el W3C . "El Consorcio World Wide Web (W3C) es una comunidad internacional que desarrolla estándares abiertos para asegurar el crecimiento a largo plazo de la Web".

¿Qué NO es HTML?

HTML se diferencia de un lenguaje de programación, porque no define el COMPORTAMIENTO (lógica) de las páginas web.

HTML utiliza "marcas" para etiquetar texto, imágenes y otro contenido para mostrarlo en un navegador Web. Las marcas HTML incluyen "elementos" especiales como <head>, <title>, <body>, <header>, <footer>, <article>, <section>, <p>, <div>, <span>, < img>, <aside>, <audio>, <canvas>, <datalist>, <details>, <embed>, <nav>, <output>, <progress>, <video>, <ul>, <ol> , <li> y muchos otros.

Es decir, con el lenguaje HTML en sí mismo, sólo podremos presentar texto e imágenes básicas en el navegador web. Para la lógica y la presentación estética

de una página web, utilizaremos otros lenguajes complementarios al HTML, tales como Javascript y CSS.

Un elemento HTML se distingue de otro texto en un documento mediante "etiquetas", que consiste en el nombre del elemento rodeado por "<" y ">". El nombre de un elemento dentro de una etiqueta no distingue entre mayúsculas y minúsculas. Es decir, se puede escribir en mayúsculas, minúsculas o una mezcla. Por ejemplo, la etiqueta <title> se puede escribir como <Title>, <TITLE> o de cualquier otra forma.

Las etiquetas vienen de a pares, la primera etiqueta del par se llama etiqueta o tag de apertura y se ve como esta: <p>. El tag o etiqueta de cierre se escribe igual que el de apertura, pero con la barra de división y se ve como esto: </p>.
Si quieres especificar que se trata de un párrafo, podrías encerrar el texto con la etiqueta de párrafo (<p>)

Veamos un ejemplo y su anatomía en la siguiente imagen:



Las partes principales del elemento son:

1. **La etiqueta de apertura**: consiste en el nombre del elemento (en este caso p), encerrado por paréntesis angulares (<>) de apertura y cierre. Establece dónde comienza o comienza dónde a tener efecto el elemento —en este caso, dónde es el comienzo del párrafo—.
2. **La etiqueta de cierre**: es igual que la etiqueta de apertura, excepto que incluye una barra de cierre (/) antes del nombre de la etiqueta. Establece dónde termina el elemento —en este caso dónde termina el párrafo—.
3. **El contenido**: este es el contenido del elemento, que en este caso es sólo texto.
4. **El elemento**: la etiqueta de apertura, más la etiqueta de cierre, más el contenido equivalente al elemento.

Los elementos y las etiquetas no son las mismas cosas. Las etiquetas comienzan o terminan un elemento en el código fuente, mientras que los elementos son parte del DOM (Document Object Model) tema que abordaremos más adelante en detalle.



Los **contenidos** contienen información adicional acerca del elemento, el cual no desea que aparezca en el contenido real del elemento. Aquí **clase** es el nombre del atributo e **"importante"** es el valor del atributo. En este caso, el atributo class permite darle al elemento un nombre identificativo, que se puede utilizar luego para encontrarlo dentro de toda la página y poder aplicarle estilos entre otras cosas.

Un atributo debe tener siempre:

1. Un espacio entre este y el nombre del elemento (o del atributo previo, si el elemento ya posee uno o más atributos).
2. El nombre del atributo, seguido por un signo de igual (=).
3. Comillas de apertura y de cierre, encerrando el valor del atributo.

Los atributos siempre se incluyen en la etiqueta de apertura de un elemento, nunca en la de cierre.

Elementos anidar

Puedes también colocar elementos dentro de otros elementos —esto se llama anidamiento—. Si, por ejemplo, quieres resaltar una palabra del texto (en el ejemplo la palabra «muy»), podemos encerrarla en un elemento <strong>, que significa que dicha palabra se debe enfatizar:

```
<p>Mi gato es <strong>muy</strong> gruñon.</p>
```

Los elementos deben abrirse y cerrarse ordenadamente, de forma tal que se encuentren claramente.

Te mostramos el mismo ejemplo pero con los elementos mal anidados, en el ejemplo de abajo, La etiqueta de apertura del elemento <p> primero, luego la del elemento <strong>, por lo tanto, debes cerrar esta etiqueta primero, y luego la de <p>. Esto es incorrecto:

```
<p>Mi gato es <strong>muy gruñon.</p></strong>
```

Si esto te llega a pasar, te vas a dar cuenta porque en el navegador hace cosas raras y no lo que vos esperabas ver. También el editor de código HTML que utilizas te avisará que hay algo incorrecto.

Comentarios HTML:

Las etiquetas de comentario son usadas para insertar comentarios en el código HTML.

Ejemplo:

```
<!-- Escribe tu comentario aquí -->

<!-- Escribe tu comentario aquí
Esto tambien funciona para
varias lineas
-->
```

Observemos que el signo de exclamación de la etiqueta de apertura no está presente en el de cierre. Los comentarios no son mostrados en el navegador web, sirven de ayuda para documentar el código fuente dentro del mismo documento HTML.

Atributos

Hemos visto algunas etiquetas HTML y su uso, como por ejemplo las etiquetas de encabezado **<h1>, <h2>,** etiqueta de párrafo **<p>** y otras etiquetas. Las etiquetas HTML también pueden tener atributos, que son bits adicionales de información.

Un atributo se utiliza para definir las características de un elemento HTML y se coloca dentro de la etiqueta de apertura del elemento. Todos los atributos se componen de dos partes: un **nombre** y un **valor**

- El nombre es la propiedad que desea establecer. Por ejemplo, el elemento de párrafo <**p>** en el ejemplo lleva un atributo cuyo nombre es **align**, que puede usar para indicar la alineación del párrafo en la página.
- El valor es lo que desea que se establezca como valor de la propiedad y siempre entre comillas. El siguiente ejemplo muestra tres valores posibles del atributo de alineación: **left, center** y **right** .

Los nombres y valores de los atributos no se distinguen entre mayúsculas y minúsculas. Sin embargo, el Consorcio World Wide Web (W3C) recomienda atributos / valores de atributo en minúsculas en su recomendación HTML 4.

```
<!DOCTYPE html>
<html>

   <head>
      <title>Align Attribute  Example</title>
   </head>

   <body>
      <p align = "left">This is left aligned</p>
      <p align = "center">This is center aligned</p>
      <p align = "right">This is right aligned</p>
   </body>

</html>
```

¿Dónde escribo el HTML?

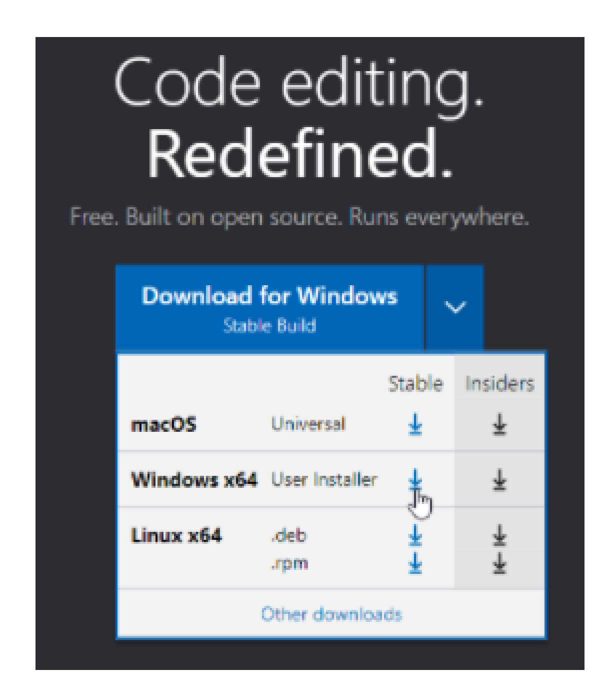
Antes de continuar con el aprendizaje de HTML, vamos a hablar un poco de **los editores de código**.

Los editores de código son programas muy sencillos que requieren poco espacio de almacenamiento y no mucho rendimiento del sistema. El dispositivo y el sistema operativo donde se utiliza básicamente depende de los gustos y necesidades del usuario. Al igual que otros tipos de software, estos programas se clasifican en software propietario, de código abierto (open source) y freeware (libre de pago), esto podría condicionar nuestra elección por uno u otro editor de código.

Existen muchos editores de código, pero aquí sólo nombraremos algunos, como, por ejemplo, Visual Studio Code, Sublime Text, Notepad + + y Atom, entre otros. Además del editor que puedes descargar y utilizar desde tu computadora, también existen editores en línea.

Nosotros nos centraremos y utilizaremos para los ejemplos del editor de código a Visual Studio Code. Este editor de código es gratuito, estable, open source, robusto y posee una buena velocidad de trabajo. Estas son algunas de las principales características de Visual Studio Code que es desarrollado y mantenido por Microsoft. A continuación, veremos brevemente cómo instalarlo:

1.- Desde la página oficial de Visual Studio Code , descargaremos el archivo para la instalación según el sistema operativo que utilicemos. En este caso mostraremos cómo hacerlo en Windows.

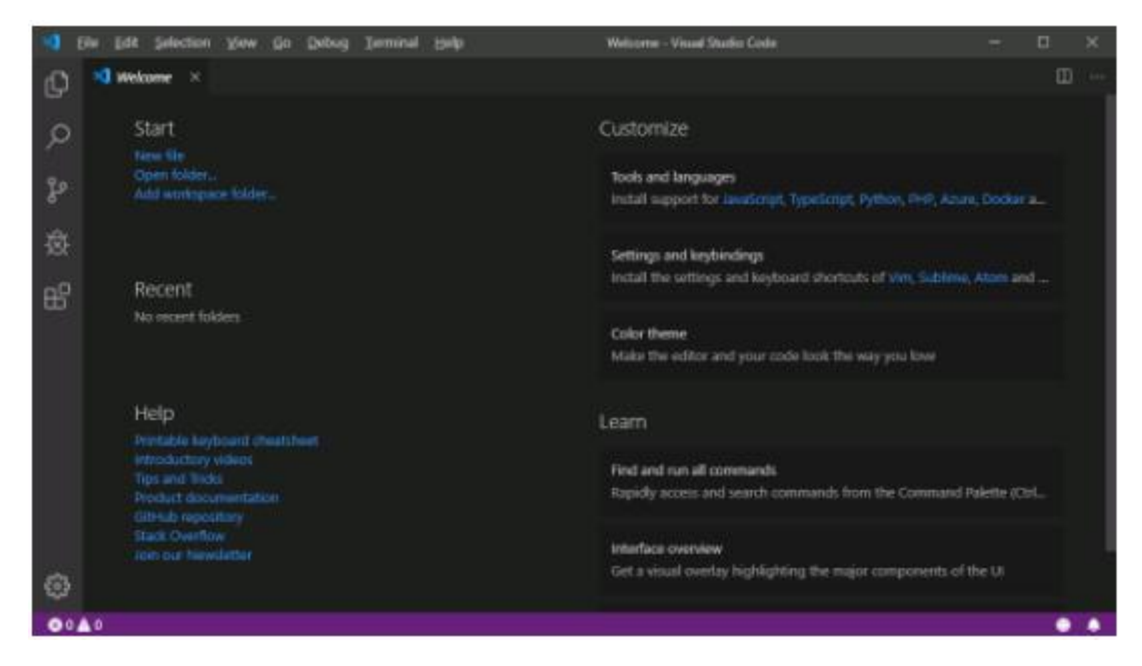


2.- Una vez descargado el archivo, hacemos clic en la barra inferior a la izquierda, donde se descargó el archivo .exe y ejecutamos el mismo.



3.- Se abrirá un cuadro de diálogo y procederemos a la instalación hasta llegar a la finalización de la misma.

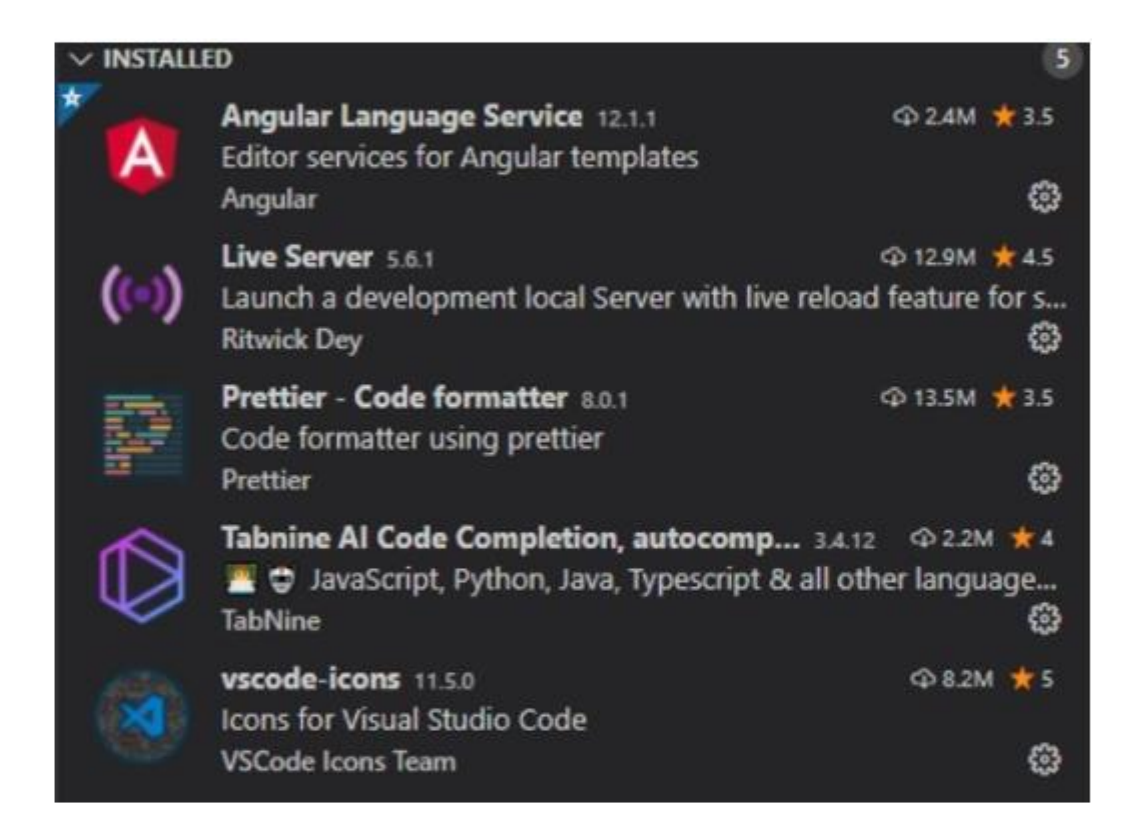
4.- Luego, una vez instalado procederemos a abrir la aplicación:



Podemos hacer clic en extensiones del menú lateral izquierdo, para agregar las extensiones que sean de nuestro agrado y nos ayuden como complemento a la hora de escribir o editar código.



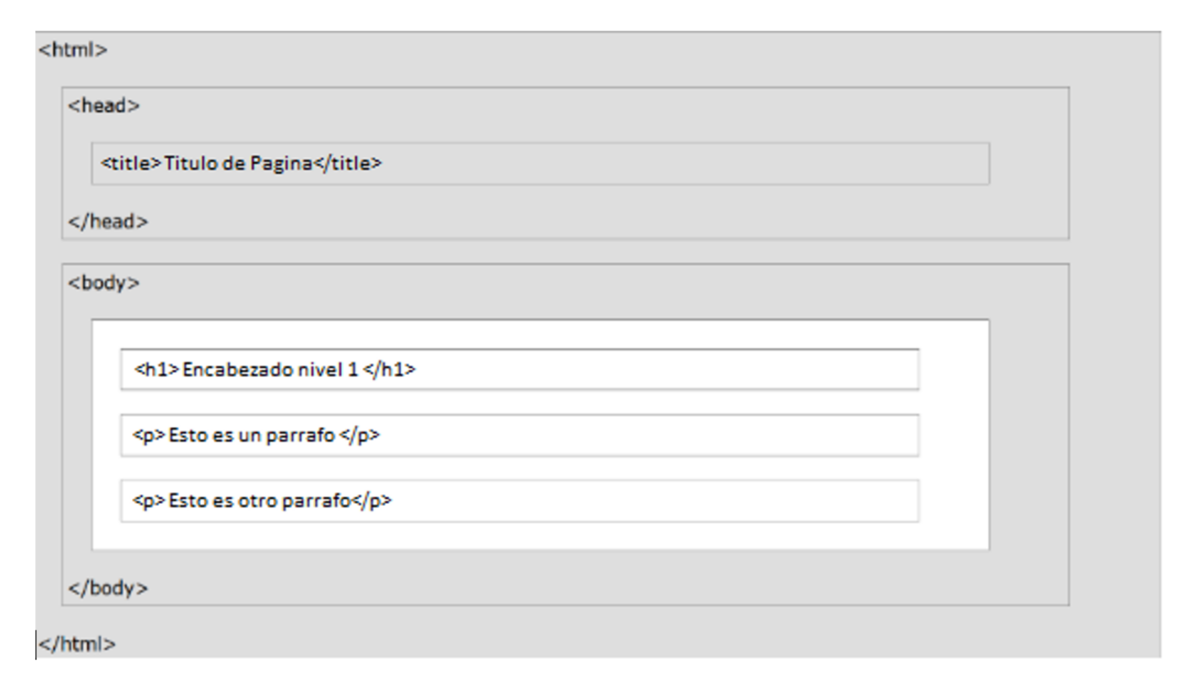
Por ejemplo, podremos instalar las siguientes extensiones:



Estructura documento HTML

Estructura básica de una página HTML

La siguiente figura es una visualización abstracta de la estructura de una página HTML. En ella podemos observar lo que representa cada etiqueta y el contenido de texto que mostraría cada elemento. También podemos visualizar mejor la analogía con la estructura de cajas de cartón. Podríamos decir con exactitud que la caja <html> contiene en su totalidad al resto de las cajas.



Ahora veamos esa estructura escrita en el código:

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Mi pagina de prueba</title>
  </head>
  <body>
    <img src="images/firefox-icon.png" alt="Mi imagen de prueba">
    <h1>Encabezado nivel 1</h1>
    <p>Esto es un parrafo</p>
    <p>Esto es otro parrafo</p>
  </body>
</html>
```

Observamos:

- **<! DOCTYPE html>** - el tipo de documento. Es un preámbulo requerido. Anteriormente, cuando HTML era joven (cerca de 1991/2), los tipos de documentos actuaban como vínculos a un conjunto de reglas que el código HTML de la página debía seguir para ser considerado bueno, lo que podía significar la verificación automática de errores y algunas otras cosas de utilidad. Sin embargo, hoy día es simplemente un artefacto antiguo que a nadie le importa, pero que debe ser incluido para que todo funcione correctamente.

- **<html> </ html** > - el elemento <html>. Este elemento encierra todo el contenido de la página entera y, a veces, se conoce como el elemento raíz.
- **<head> </head>** - el elemento <head>. Este elemento actúa como un contenedor de todo aquello que desea incluir en la página HTML que no es contenido visible por los visitantes de la página. Incluye cosas como palabras clave (keywords), una descripción de la página que quieres que aparezca en resultados de búsquedas, código CSS para dar estilo al contenido, declaraciones del juego de caracteres, etc.
- **<meta charset = "utf-8">** - <meta>. Este elemento establece el juego de caracteres que tu documento usará en utf-8, que incluye casi todos los caracteres de todos los idiomas humanos. Básicamente, puede manejar cualquier contenido de texto que pueda incluir. No hay razón para no establecerlo, y puede evitar problemas en el futuro.
- **<title> </ title > -** el elemento <title> establece el título de tu página, que es el título que aparece en la pestaña o en la barra de título del navegador cuando la página es cargada, y se usa para describir la página cuando es añadida a los marcadores o como favorita.
- **<body> </body>** - el elemento <body>. Encierra todo el contenido que deseas mostrar a los usuarios web que visitan tu página, ya sea texto, imágenes, videos, juegos, pistas de audio reproducibles, y demás.
- **<h1>,…, <h6>** Los elementos de encabezado implementan seis niveles de encabezado del documento, <h1> es el más importante, y <h6>, el menos importante. Un elemento de encabezado describe brevemente el tema de la sección que presenta. La información de encabezado puede ser usada por los agentes usuarios, por ejemplo, para construir una tabla de contenidos para un documento automáticamente. Sus etiquetas son <h1>,…, <h6> y </h1>,…, </h6>.
- **<p>** El elemento <p> (párrafo) es apropiado para distribuir el texto en párrafos. Sus etiquetas son <p> y </p>.
- **** El elemento HTML indica que el texto debe ser representado con una variable bold, o negrita, de la tipografía que se esté usando. Sin darle al texto importancia adicional. Sus etiquetas son y .
- **** El elemento destaca el texto. Sus etiquetas son y . El elemento <strong> le da al texto más énfasis que el elemento <b>, con una importancia más alta semánticamente.
- **<i>** El elemento HTML <i> muestra el texto marcado con un estilo en cursiva o itálica. Sus etiquetas son <i> e </i>.
- **** El elemento HTML es apropiado para marcar con énfasis en el texto. El elemento puede ser anidado, con cada nivel de anidamiento indicando un mayor grado de énfasis. Sus etiquetas son y .
- **
** El elemento HTML
 produce un salto de línea en el texto (retorno de carro). Es útil para escribir un poema o una dirección, donde la división de las líneas es significante. No lo utilices para incrementar el espacio entre líneas de texto; para ello utiliza la propiedad margin de CSS o el elemento <p>.
- **
** El elemento HTML <li> o elemento de lista declara cada uno de los elementos de una lista. Sus etiquetas son <li> e .

- **
** El elemento <ol> permite definir listas o viñetas ordenadas con numeración o alfabéticamente. Sus etiquetas son <ol> y .
- **
** El elemento HTML <ul> de "lista desordenada" - lista no ordenada. crea una lista no ordenada. Sus etiquetas son <ul> y .
- **<div>** El elemento HTML <div> es exclusivamente usado como contenedor para otros elementos HTML. En conjunto con CSS, el elemento <div> puede ser usado para agregar formato a un bloque de contenido. Sus etiquetas son <div> y </div>.
- **<img>** El elemento HTML <img> posee los atributos src y alt pero no tiene etiqueta de cierre. Se puede representar así <img src = "imagen.png" alt = "Mi descripción de imagen"> Un elemento <img> no encierra contenido. También a este tipo de elemento se lo conoce como elemento vacío. El propósito del elemento <img> es desplegar una imagen en la página web, en el lugar que corresponde según la estructura del documento.
- El nombre de archivo de la imagen de origen está especificado por el atributo src. Los navegadores web no siempre muestran la imagen a la que el elemento hace referencia. Es el caso de los navegadores no gráficos (incluidos aquellos usados por personas con problemas de visión), si el usuario elige no mostrar la imagen, o si el navegador es incapaz de mostrarla porque no es válida o soportada. En ese caso, el navegador la reemplazará con el texto definido en el atributo alt
- **<a>** El Elemento HTML Anchor <a> crea un enlace a otras páginas de Internet, archivos o ubicaciones dentro de la misma página, direcciones de correo, o cualquier otra URL que especifiquemos en sus atributos. Se puede representar así   donde la dirección del enlace está especificada por el atributo href.
- Dentro del atributo href la URL puede escribirse de forma absoluta (incluyendo el dominio) o relativa (sin incluir el dominio) solo para enlaces dentro del mismo dominio. Tanto de una forma u otra, la ruta de carpetas debe especificarse.

Siguiendo con la descripción del atributo href del elemento <a>, podemos dividir los enlaces o links en 3 tipos:

1. Enlaces internos: son los que se dan entre páginas web del mismo dominio.
2. Enlaces externos: son los que se dan entre páginas web de distinto dominio.
3. Enlaces de posición (o marcadores):
    - De un lugar a otro dentro de la misma página.
    - De un lugar a otro lugar concreto de otra página del mismo dominio.
    - De un lugar a otro lugar concreto de una página de otro dominio.

Encabezados

Etiquetas de encabezado

Cualquier documento comienza con un encabezado. Puede utilizar diferentes tamaños para sus títulos. HTML también tiene seis niveles de encabezados, que utilizan los elementos <h1>, <h2>, <h3>, <h4>, <h5> y <h6>. Mientras muestra cualquier encabezado, el navegador agrega una línea antes y una línea después de ese encabezado.

```
<!DOCTYPE html>
<html>

   <head>
      <title>Heading Example</title>
   </head>

   <body>
      <h1>This is heading 1</h1>
      <h2>This is heading 2</h2>
      <h3>This is heading 3</h3>
      <h4>This is heading 4</h4>
      <h5>This is heading 5</h5>
      <h6>This is heading 6</h6>
   </body>

</html>
```

Formato

Si usas un procesador de texto, debes estar familiarizado con la capacidad de hacer que el texto esté en **negrita**, *cursiva* o <u>subrayado</u>; estas son solo tres de las diez opciones disponibles para indicar cómo puede aparecer el texto en HTML y XHTML:

Texto en negrita

Todo lo que aparece dentro del elemento ** ... ** se muestra en negrita

Texto en cursiva

Todo lo que aparece dentro del elemento **<i> ... </i>** se muestra en cursiva

<u>**Texto subrayado**</u>

Todo lo que aparece dentro del elemento **<u> ... </u>** se muestra subrayado

~~Texto de tachado~~

Todo lo que aparece dentro del elemento **<strike> ... </strike>** se muestra tachado

Fuente monoespaciada

El contenido de un elemento **<tt> ... </tt>** está escrito en fuente monoespaciada. La mayoría de las fuentes se conocen como fuentes de ancho variable porque diferentes letras tienen diferentes anchos (por ejemplo, la letra 'm' es más ancha que la letra 'i'). Sin embargo, en una fuente monoespaciada, cada letra tiene el mismo ancho.

Meta Tags

HTML te permite especificar metadatos: información adicional importante sobre un documento de diversas formas. Los elementos META se pueden utilizar para incluir pares de nombre - valor que describen las propiedades del documento HTML, como el autor, la fecha de caducidad, una lista de palabras clave, el autor del documento, etc.

La etiqueta **<meta>** se utiliza para proporcionar dicha información adicional. Esta etiqueta es un elemento vacío y, por lo tanto, no tiene una etiqueta de cierre, pero lleva información dentro de sus atributos.

Puede incluir una o más meta etiquetas en su documento en función de la información que desee mantener, pero en general, las meta etiquetas no indican la apariencia física del documento, por lo que desde el punto de vista de la apariencia, no importa si se incluye o no.

listas, imágenes y enlaces HTML

A continuación desarrollamos un ejemplo en donde utilizamos los elementos <ol>, <ul>, <img> y <a>.

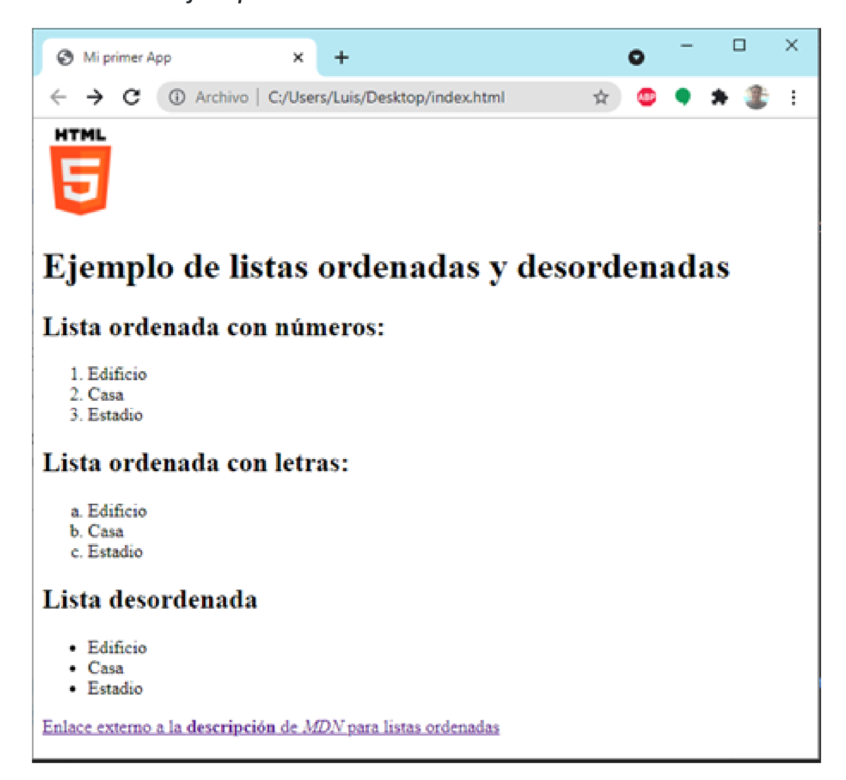
En el caso del elemento <ol> si le asignamos el atributo type = “a” le indicamos al explorador web que vamos a realizar una lista ordenada alfabéticamente en minúscula. Vale aclarar que lo que se muestra ordenado es el prefijo que se asigna al ítem y no los propios nombres de cada ítem. Si no indicamos el atributo type, por defecto, la lista sale ordenada numéricamente.

En el caso del elemento <img>, se utilizó como referencia, una imagen del sitio https://3con14.biz/html/ y se le aplicaron atributos de ancho y alto para reducir su tamaño relativo a la pantalla (los explicaremos con más profundidad en el módulo de CSS).

Con el elemento <a> insertamos un enlace a una página externa, en este caso, de Mozilla Developer Network

```
index.html > ...
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Mi primer App</title>
</head>
<body>
<img src="http://3con14.biz/html/_data/_config/html5.png" alt="Logo HTML5" width="10%" height="10%">
 <h1>Ejemplo de listas ordenadas y desordenadas</h1>
 <h3>Lista ordenada con números:</h3>
 <ol>
   <li>Edificio</li>
   <li>Casa</li>
   <li>Estadio</li>
 </ol>
 <h3>Lista ordenada con letras:</h3>
 <ol type="a">
   <li>Edificio</li>
   <li>Casa</li>
   <li>Estadio</li>
 </ol>
 <h3>Lista desordenada</h3>
 <ul>
   <li>Edificio</li>
   <li>Casa</li>
   <li>Estadio</li>
 </ul>
<a href="https://developer.mozilla.org/es/docs/Web/HTML/Element/ol">Enlace a la <strong>descripción</strong>
  de <em>MDN</em> para listas ordenadas</a>
</body>
</html>
```

Ejemplo en el IDE Visual Studio Code



Las listas en HTML son muy importantes, no solo se utilizan para listar, sino también se utilizan para el menú, cuando ingresas en una web y tiene un menú desplegable, seguramente usaron listas que tienen un estilo y comportamiento específico, pero en el fondo son listas.

En HTML tenemos 3 tipos de listas a utilizar para diferentes usos:

1. Listas Ordenadas.
2. Listas Desordenadas.
3. Lista de Definición.

Listas Ordenadas

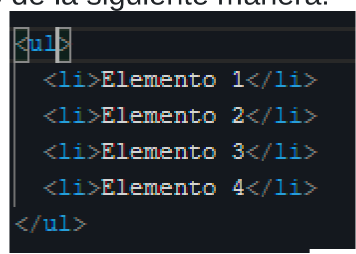
Las listas ordenadas son listas en las que el orden de los elementos **SI importa** y se indican con la etiqueta <ol> , dentro se le anida otra etiqueta <li> quedando de la siguiente manera:

```
<ol>
    <li>Elemento 1</li>
    <li>Elemento 2</li>
    <li>Elemento 3</li>
    <li>Elemento 4</li>
</ol>
```

1. Elemento 1
2. Elemento 2
3. Elemento 3
4. Elemento 4

Listas Desordenadas

Las listas desordenadas son listas en las que el orden de los elementos **NO importa** y se indican con la etiqueta <ul> , dentro, se le anida otra etiqueta <li> quedando de la siguiente manera:



```
<ul>
    <li>Elemento 1</li>
    <li>Elemento 2</li>
    <li>Elemento 3</li>
    <li>Elemento 4</li>
</ul>
```

- Elemento 1
- Elemento 2
- Elemento 3
- Elemento 4

¿Recordarás que las etiquetas pueden tener Atributos?

En las listas ordenadas **existen varios Atributos** (por ejemplo: start, type, reverse, etc) que se aplican al elemento <ol> . Investiga qué se puede hacer con esos Atributos, algunas aplican solo para las listas ordenadas y otras para listas desordenadas. Usa tus habilidades de búsqueda ya que en los ejercicios siguientes lo vas a necesitar.

Listas de Definiciones

Estas listas de definiciones no son tan comunes, por lo que te invitamos a que busques en internet cuáles son las etiquetas HTML que se utilizan para crear esta lista.

Iframe

Puedes definir un marco en línea con la etiqueta HTML **<iframe>**. La etiqueta <iframe> no está relacionada de alguna manera con la etiqueta <frameset>, sino que puede aparecer en cualquier parte de su documento. La etiqueta <iframe> define una región rectangular dentro del documento en la que el navegador puede mostrar un documento separado, incluidas las barras de desplazamiento y los bordes. Un marco en línea se utiliza para incrustar otro documento dentro del documento HTML actual.

Fuentes

Las fuentes juegan un papel muy importante para hacer que un sitio web sea más fácil de usar y aumentar la legibilidad del contenido. La cara y el color de la fuente depende completamente de la computadora y el navegador que estás utilizando para ver tu página, pero puedes usar la etiqueta HTML **** para agregar estilo, tamaño y color al texto de su sitio web. Puedes usar una etiqueta <**basefont>** para configurar todo tu texto en el mismo tamaño, cara y color.

La etiqueta de fuente tiene tres atributos llamados **tamaño, color** y **cara** para personalizar tus fuentes. Para cambiar cualquiera de los atributos de fuente en cualquier momento dentro de tu página web, simplemente usa la etiqueta . El texto que sigue permanecerá cambiado hasta que cierres con la etiqueta . Puedes cambiar uno o todos los atributos de fuente dentro de una etiqueta <font>.

Nota: La *fuente* y *BASEFONT* son etiquetas que están en desuso y se supone que será eliminado en una futura versión de HTML. Por lo tanto, no deben usarse, en su lugar se sugiere usar estilos CSS para manipular tus fuentes. Pero aún con el propósito de aprender, este capítulo explicará en detalle las etiquetas de fuente y fuente base.

Establecer tamaño de fuente

Puedes establecer el tamaño de la fuente del contenido mediante el atributo de **Tamaño**. El rango de valores aceptados es de 1 (menor) a 7 (mayor). El tamaño predeterminado de una fuente es 3

HTML5 lo nuevo

Hasta ahora hemos leído y hemos visto ejemplos sobre HTML básico, pero, en la actualidad existe en plena vigencia la versión HTML 5.
Es decir, HTML5 es la versión más reciente de HTML homologada por la W3C.

Una de las principales ventajas de HTML5 es la inclusión de elementos semánticos, o marcadores semánticos, que nos ayudan a definir las distintas

divisiones de una página web. Es decir, los elementos semánticos de HTML5, nos ayudan a identificar cada sección del documento y organizar el cuerpo de una página web de una manera eficiente y estandarizada.

En la figura siguiente, observamos el esquema básico de los elementos semánticos introducidos por la última y por la versión actual de HTML5:

<header></header>

<nav></nav>

<section>

<article></article>

<article></article>

</section>

<aside>

</aside>

<footer></footer>

Según el W3C una Web Semántica:

"*Permite que los datos sean compartidos y reutilizados en todas las aplicaciones, las empresas y las comunidades*."

¿Por qué utilizar elementos semánticos?

En HTML4 los desarrolladores usaban sus propios nombres como identificación de los elementos, tales como: cabecera, superior, inferior, pie de página, menú, navegación, principal, contenedor, contenido, artículo, barra lateral, etc., se hacía

imposible que los motores de búsqueda identificaran el contenido correcto de la página web.

Elementos semánticos = elementos con significado.

Un elemento semántico describe claramente su significado tanto para el navegador web como para el desarrollador.

- Ejemplos de elementos no semánticos: <div> y <span> - No dice nada sobre su contenido.
- Ejemplos de elementos semánticos: <form>, <table> y <article> - Definen claramente su contenido.

Descripción de elementos semánticos HTML5

Elemento HTML <section>

El elemento <section> define una sección en un documento. Según la documentación HTML del W3C: " *Una sección es una agrupación temática de contenido, normalmente con un encabezado* ".

Una página web normalmente se puede dividir en secciones para la introducción, el contenido y la información de contacto.

Elemento HTML <article>

El elemento <article> específica contenido autónomo e independiente. Un artículo debe tener sentido por sí solo y debe ser posible distribuirlo independientemente del resto del sitio web.

Ejemplos de dónde se puede utilizar un elemento <article>:

- Publicación en el foro
- Entrada en el blog
- Artículo de periódico

Elemento HTML <header>

El elemento <header> sirve para especificar contenido de tipo introductorio o un conjunto de enlaces de navegación.

Un elemento <header> normalmente contiene:

1. Uno o más elementos de encabezado (<h1> - <h6>).
2. Logo o icono.
3. Información de autoría.

Nota: puede tener varios elementos <encabezado> en un documento HTML. Sin embargo, el elemento <header> no puede ser colocado dentro de elementos <footer>, <address> o dentro de otro elemento <header>.

Elemento HTML <footer>

El elemento <footer> define un pie de página para un documento o sección.

Un elemento <footer> normalmente contiene:

1. Información de autoría.
2. Información registrada.
3. Información de contacto.
4. Mapa del sitio.
5. Volver a los enlaces superiores.
6. Documentos relacionados.

Puede tener varios elementos <footer> en un documento.

Elemento HTML <nav>

El elemento HTML <nav> representa una sección de una página cuyo propósito es proporcionar enlaces de navegación, ya sea dentro del documento actual o en otros documentos. Ejemplos comunes de secciones de navegación son: menús, tablas de contenido e índices.

Debemos tener en cuenta que NO todos los enlaces de un documento deben estar dentro de un elemento <nav>. El elemento <nav> está destinado sólo para el bloque principal de enlaces de navegación.

Los navegadores, como los lectores de pantalla para usuarios con capacidades diferentes, pueden utilizar este elemento para determinar si se debe omitir la representación inicial de este contenido.

Elemento HTML <aside>

El elemento HTML <aside> representa una sección de una página que consiste en contenido que está indirectamente relacionado con el contenido principal del documento. Estas secciones son a menudo representadas como barras laterales o como inserciones y contienen una explicación al margen como una definición de glosario, elementos relacionados indirectamente, como publicidad, la biografía del autor, o en aplicaciones web, la información de perfil o enlaces a blogs relacionados.

Colores

Los colores son muy importantes para darle una buena apariencia a tu sitio web. Puedes especificar colores a nivel de página usando la etiqueta <body> o puedes establecer colores para etiquetas individuales usando el atributo **bgcolor**.

La etiqueta <body> tiene los siguientes atributos que se pueden usar para establecer diferentes colores:

- **bgcolor**: establece un color para el fondo de la página.
- **text**: establece un color para el cuerpo del texto.
- **alink**: establece un color para los enlaces activos o los enlaces seleccionados.
- **link**: establece un color para el texto vinculado.
- **vlink**: establece un color para los enlaces visitados, es decir, para el texto vinculado en el que ya ha hecho clic.

Métodos de codificación de colores HTML

Existen tres métodos diferentes para configurar los colores en tu página web:

- **Nombres de colores**: puede especificar nombres de colores directamente como verde, azul o rojo.
- **Códigos hexadecimales**: un código de seis dígitos que representa la cantidad de rojo, verde y azul que compone el color.
- **Valores decimales o porcentuales de color**: este valor se especifica mediante la propiedad rgb ().

Ahora veremos estos esquemas de coloración uno por uno.

Colores HTML – Nombres de colores

Puedes especificar un nombre de color directo para establecer el texto o el color de fondo. W3C ha enumerado 16 nombres de colores básicos que se validarán con un validador HTML, pero hay más de 200 nombres de colores diferentes compatibles con los principales navegadores.

Estándar W3C 16 colores

Aquí está la lista de nombres de 16 colores estándar del W3C y se recomienda usarlos.



Introducción CSS

El gran impulso de los lenguajes de hojas de estilos se produjo con el boom de Internet y el crecimiento exponencial del lenguaje HTML. Entre la fuerte competencia de navegadores web y la falta de un estándar para la definición de los estilos, se dificultaba la creación de documentos con la misma apariencia en diferentes navegadores.

A mediados de la década de 1990 el W3C, encargado de crear todos los estándares relacionados con la web, propuso la creación de un lenguaje de hojas de estilos específicos para el lenguaje HTML y se presentaron propuestas. Las dos propuestas que se tuvieron en cuenta fueron la CHSS (Cascading HTML Style Sheets) y la SSP (Stream-based Style Sheet Proposal). La propuesta CHSS fue realizada por Håkon Wium Lie y SSP fue propuesto por Bert Bos . Entre finales de 1994 y 1995 Lie y Bos se unieron para definir un nuevo lenguaje que tomaba lo mejor de cada propuesta y lo llamaron CSS , Cascading Style Sheets.

En 1995, el W3C decidió gestionar el desarrollo y estandarización de CSS y agregó el proyecto a su grupo de trabajo de HTML. A finales de 1996, el W3C publicó la primera recomendación oficial, conocida como " CSS nivel 1 ".

¿Qué es CSS?.

Css es un lenguaje que trabaja junto con HTML para proporcionar estilos visuales a los elementos de un documento web.

Características:

- Ahorra trabajo. Se puede controlar el diseño de varias páginas HTML a la vez.
- Se pueden almacenar en archivos * .css

CSS 3 es la última versión estándar.

¿Para qué utilizar CSS?

Para definir estilos en los documentos web, incluyendo el diseño, la disposición de los elementos y para responder a las variaciones en la pantalla en cuanto a diferentes dispositivos y tamaños de pantalla.

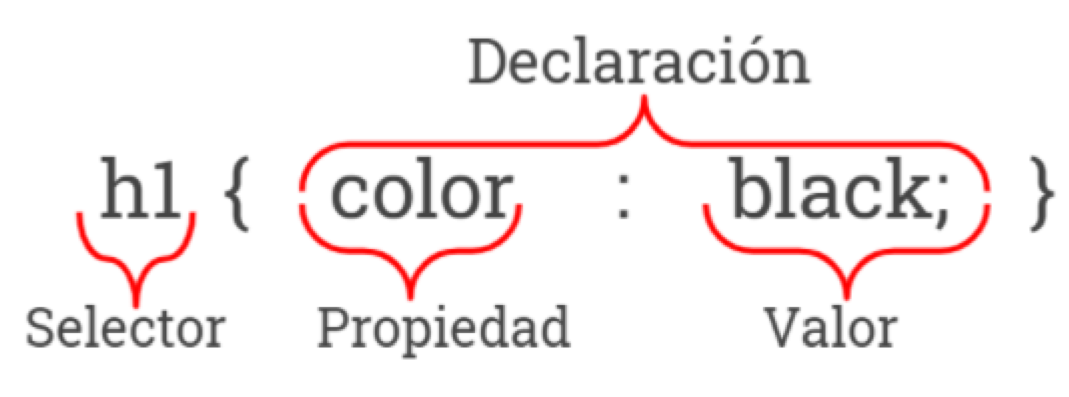
Ventajas

1. Control centralizado de la presentación de un sitio web completo con lo que se agiliza considerablemente la actualización y mantenimiento.

2. Los Navegadores permiten a los usuarios especificar su propia hoja de estilo local que será aplicada a un sitio web, con lo que aumenta considerablemente la accesibilidad. Por ejemplo, las personas con deficiencias visuales pueden configurar su propia hoja de estilo para aumentar el tamaño del texto o remarcar más los enlaces.
3. Una página puede disponer de diferentes hojas de estilo según el dispositivo que la muestra o incluso una elección del usuario. Por ejemplo, para ser impresa, mostrada en un dispositivo móvil, o ser "leída" por un sintetizador de voz.
4. El documento HTML en sí mismo es más claro de entender y se consigue reducir considerablemente su tamaño (siempre y cuando no se utiliza estilo en línea).

Reglas CSS

CSS define un conjunto de reglas que permiten describir cada una de las partes que componen los estilos CSS.



Sintaxis CSS

Selector: indica el elemento o elementos HTML a los que se aplica la regla CSS.

Declaración: especifica los estilos que se aplican a los elementos.

Propiedad: permite modificar el aspecto de una característica del elemento.

Valor: indica el nuevo valor de la característica modificada en el elemento.

Selectores

Nos indican qué elemento HTML hay que aplicar el estilo. Una misma regla puede aplicar a varios selectores y, a un mismo selector se le pueden aplicar varias reglas.

Selectores básicos

Selector universal: se utiliza para seleccionar todos los elementos de la página:

```
* {
    margin: 0;
    padding: 0;
    }
```

Selector de tipo o etiqueta: selecciona todos los elementos de la página cuya etiqueta HTML coincide con el valor del selector:

```
h1 {
    color: red;
    }

h2 {
    color: blue;
    }

p {
    color: black;
    }
```

Selector descendente

Selecciona los elementos que se encuentran dentro de otros elementos. Un elemento es descendente de otro cuando se encuentra entre las etiquetas de apertura y de cierre de otro elemento.

```
p span { color: red; }
h1 span { color: blue; }
```

La sintaxis formal del selector descendente se muestra a continuación:

elemento1 elemento2 elemento3 ... elementoN

```
p span { color: green; }
```

Si el código HTML de la página es el siguiente:

```
<p>
  Ejemplo...<span>texto1</span>...
  <a href="https://www..../">Ejemplo...<span>texto2</span></a>
</p>
```

El selector p span selecciona tanto texto1 como texto2. El motivo es que, en el selector descendente, un elemento no tiene que ser descendiente directo del otro. La única condición es que un elemento debe estar dentro de otro elemento, sin importar el nivel de profundidad en el que se encuentre.

No se les aplica la regla CSS anterior si el resto de los elementos <span> de la página no están dentro de un elemento <p>.

Los selectores descendentes nos permiten aumentar la precisión del selector de tipo o etiqueta. Así, utilizando el selector descendente podemos aplicar diferentes estilos a los elementos del mismo tipo.

Selector de clase o selector Class

Si consideramos el siguiente código HTML de ejemplo:

```
<body>
  <p>Lorem ipsum dolor sit amet...</p>
  <p>Nunc sed lacus et est adipiscing accumsan...</p>
  <p>Class aptent taciti sociosqu ad litora...</p>
</body>
```

¿Cómo podemos aplicar estilos CSS sólo al primer párrafo?

El selector universal (*) no se puede utilizar porque seleccionaría todos los elementos de la página. El selector de tipo o etiqueta (p) tampoco se puede utilizar porque seleccionaría todos los párrafos. Por último, el selector descendente (body p) tampoco se puede utilizar porque todos los párrafos se encuentran en el mismo sitio.

Una de las soluciones más sencillas para aplicar estilos a un solo elemento de la página consiste en utilizar el atributo class de HTML sobre ese elemento para indicar directamente la regla CSS que se le debe aplicar:

```
<body>
  <p class="destacado">Lorem ipsum dolor sit amet...</p>
  <p>Nunc sed lacus et est adipiscing accumsan...</p>
  <p>Class aptent taciti sociosqu ad litora...</p>
</body>
```

A continuación, creamos en el archivo CSS una nueva regla llamada destacado con todos los estilos que vamos a aplicar al elemento. Para que el navegador no confunda este selector con los otros tipos de selectores, prefijamos el valor del atributo class con un punto (.) Tal como vemos en la siguiente línea:

```
.destacado { color: blue; }
```

El selector destacado se interpreta como "cualquier elemento de la página cuyo atributo class sea igual a destacado", por lo que solamente el primer párrafo del anterior ejemplo cumple esa condición.

Este tipo de selectores de clase son los más utilizados junto con los selectores de ID que veremos a continuación. La principal característica de este selector es que en una misma página HTML varios elementos diferentes pueden utilizar el mismo valor en el atributo class:

```
<body>
  <p class="destacado">Lorem ipsum dolor sit amet...</p>
  <p>Nunc sed lacus et <a href="#" class="destacado">est adipiscing</a> accumsan...</p>
  <p>Class aptent taciti <em class="destacado">sociosqu ad</em> litora...</p>
</body>
```

Los selectores de clase resultan imprescindibles para diseñar páginas web complejas, ya que nos permiten disponer de una precisión total al seleccionar los elementos. Además, estos selectores nos permiten reutilizar los mismos estilos para varios elementos diferentes:

```
.aviso {
    padding: 0.5em;
    border: 1px solid #98be10;
    background: #f6feda;
  }

  .error {
    color: #930;
    font-weight: bold;
  }
```

```
<span class="error">...</span>

<div class="aviso">...</div>
```

El elemento <span> tiene un atributo class = "error", por lo que se le aplica las reglas CSS indicadas por el selector .error. Por su parte, el elemento <div> tiene un atributo class = "aviso", por lo que su estilo es el que define las reglas CSS del selector .aviso.

En ciertos casos, es necesario restringir el alcance del selector de clase. Si utilizamos el ejemplo anterior:

```
<body>
  <p class="destacado">Lorem ipsum dolor sit amet...</p>
  <p>Nunc sed lacus et <a href="#" class="destacado">est adipiscing</a> accumsan...</p>
  <p>Class aptent taciti <em class="destacado">sociosqu ad</em> litora...</p>
</body>
```

¿Cómo podemos aplicar estilos solamente al párrafo cuyo atributo class sea igual a destacado?

Combinando el selector de tipo y el selector de clase, se obtiene un selector mucho más específico:

```
p.destacado { color: orange }
```

Interpretamos al selector p.destacado como "aquellos elementos de tipo <p> que dispongan de un atributo class con valor destacado". De la misma forma, el selector a.destacado solamente seleccionaría los enlaces cuyo atributo class sea igual a destacado.

De lo anterior deducimos que el atributo .destacado es equivalente a * .destacado, por lo que todos los diseñadores obvian el símbolo * al escribir un selector de clase normal.

No debemos confundir el selector de clase con los selectores anteriores:

/ * Todos los elementos de tipo "p" con atributo class = "aviso" * /

p.aviso {...}

/ * Todos los elementos con atributo class = "aviso" que estén dentro

de cualquier elemento de tipo "p" * /

p .aviso {...} / * notar el espacio entre py la clase * /

/ * Todos los elementos "p" de la página y todos los elementos con

atributo class = "aviso" de la página * /

p, .aviso {...} / * notar la, (coma) entre py la clase * /

Para completar, es posible aplicar los estilos de varias clases CSS sobre un mismo elemento. La sintaxis es similar, pero los diferentes valores del atributo class se separan con espacios en blanco.

```
<p class="especial destacado error">Párrafo de texto...</p>
```

Al párrafo anterior le aplicamos los estilos definidos en las reglas .especial, .destacado y .error, por lo que en el siguiente ejemplo, el texto del párrafo se vería de color rojo, en negrita y con un tamaño de letra de 15 píxel:

```
.error { color: red; }
.destacado { font-size: 15px; }
.especial  { font-weight: bold; }
```

```
<p class="especial destacado error">Párrafo de texto...</p>
```

Si un elemento dispone de un atributo class con más de un valor, es posible utilizar un selector más avanzado:

```
.error { color: red; }
.error.destacado { color: blue; }
.destacado { font-size: 15px; }
.especial  { font-weight: bold; }
```

```
<p class="especial destacado error">Párrafo de texto...</p>
```

En el ejemplo anterior, el color de la letra del texto es azul y no rojo. Esto se debe a que hemos utilizado un selector de clase múltiple .error.destacado, que se interpreta como "aquellos elementos de la página que están disponibles de un atributo class con al menos los valores error y destacado".

Selectores de ID

En ocasiones, es necesario aplicar estilos CSS a un único elemento de la página. Aunque puede aplicar un selector de clase para un único elemento, existe otro selector más eficiente en este caso.

El selector de ID permite seleccionar un elemento de la página a través del valor de su atributo id. Este tipo de selectores sólo selecciona un elemento de la página porque el valor del atributo id no se puede repetir en dos elementos diferentes de una misma página.

La sintaxis de los selectores de ID es muy parecida a la de los selectores de clase, salvo que se utiliza el símbolo de la almohadilla (#) en vez del punto (.) Como prefijo del nombre de la regla CSS:

```
#destacado { color: red; }
```

```
<p>Primer párrafo</p>
<p id="destacado">Segundo párrafo</p>
<p>Tercer párrafo</p>
```

En el ejemplo anterior, el selector #destacado solamente selecciona el segundo párrafo (cuyo atributo id es igual a destacado).

La principal diferencia entre este tipo de selector y el selector de clase tiene que ver con HTML y no con CSS. En una misma página, el valor del atributo id debe ser único, de forma que dos elementos diferentes no pueden tener el mismo valor de id. Sin embargo, el atributo class no es obligatorio que sea único, de forma que muchos elementos HTML diferentes pueden compartir el mismo valor para su atributo class.

De esta forma, la recomendación general es la de utilizar el selector de ID cuando se quiere aplicar un estilo a un solo elemento específico de la página y utilizar el selector de clase cuando queremos aplicar un estilo determinado a varios elementos diferentes de la página HTML.

Al igual que los selectores de clase, en este caso también podemos restringir el alcance del selector mediante la combinación con otros selectores. El siguiente ejemplo aplica la regla CSS solamente al elemento de tipo <p> que tenga un atributo id igual al indicado:

```
p#aviso { color: blue; }
```

A primera vista, restringir el alcance de un selector de ID puede parecer absurdo. En realidad, un selector de tipo p # aviso sólo tiene sentido cuando el archivo CSS se aplica sobre muchas páginas HTML diferentes.

En este caso, algunas páginas pueden disponer de elementos con un atributo id igual a aviso y que no sean párrafos, por lo que la regla anterior no se aplica sobre esos elementos.

No debe confundirse el selector de ID con los selectores anteriores:

/ * Todos los elementos de tipo "p" con atributo id = "aviso" * /

p # aviso {...}

/ * Todos los elementos con atributo id = "aviso" que estén dentro

de cualquier elemento de tipo "p" * /

p #aviso {...}

/ * Todos los elementos "p" de la página y todos los elementos con

atributo id = "aviso" de la página * /

p, #aviso {...}

Selectores avanzados

Se trata de un selector similar al selector descendente, pero muy diferente en su funcionamiento. Se utiliza para seleccionar un elemento que es hijo directo de otro elemento y se indica mediante el "signo de mayor que" (>):

```
p > span { color: blue; }
```

```
<p><span>Texto1</span></p>
<p><a href="#"><span>Texto2</span></a></p>
```

En el ejemplo anterior, el selector p> span se interpreta como "cualquier elemento <span> que sea hijo directo de un elemento <p>", por lo que el primer elemento <span> cumple la condición del selector. Sin embargo, el segundo elemento <span> no la cumple porque es descendiente pero no es hijo directo de un elemento <p>.

El siguiente ejemplo muestra las diferencias entre el selector descendente y el selector de hijos:

```
p a { color: red; }
p > a { color: red; }
```

```
<p><a href="#">Enlace1</a></p>
<p><span><a href="#">Enlace2</a></span></p>
```

El primer selector es de tipo descendente y por tanto se aplica a todos los elementos <a> que se encuentran dentro de los elementos <p>. En este caso, los estilos de este selector aplican se a los dos enlaces.

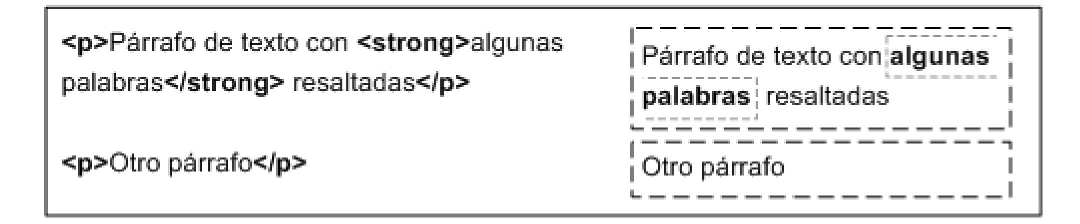
Por otra parte, el selector de hijos obliga a que el elemento <a> sea hijo directo de un elemento <p>. Por lo tanto, los estilos del selector p> a no se aplican al segundo enlace del ejemplo anterior.

Modelo de Cajas

Tal como dijimos en la unidad de HTML, si hacemos una analogía con una estructura de cajas de cartón, podemos decir que hay ciertas cajas que van dentro de otras y ciertas cajas que van una al lado de otra.

El "modelo de caja" es probablemente la característica más importante del lenguaje de hojas de estilos CSS, ya que condiciona el diseño de todas las páginas web. El modelo de cajas es el comportamiento de CSS que hace que todos los elementos de las páginas sean representados mediante cajas rectangulares.

Cada vez que se inserta un elemento HTML, se crea una nueva caja rectangular que encierra los contenidos de ese elemento. La siguiente imagen muestra tres cajas rectangulares que crean los tres elementos HTML de una porción de página de ejemplo.



Referencia: uniwebsidad

Los navegadores web crean y colocan las cajas de forma automática, pero CSS permite modificar todas sus características. Cada una de las cajas está formada por seis partes que se describen a continuación:

1. **Content** (contenido): se trata del contenido HTML del elemento (las palabras de un párrafo, una imagen, el texto de una lista de elementos, etc.)
2. **Padding** (relleno): espacio libre opcional existente entre el contenido y el borde.
3. **Border** (borde): línea que encierra completamente el contenido y su relleno.
4. **Background-imagen** (Imagen de fondo): imagen que se muestra por detrás del contenido y el espacio de relleno.
5. **Background-color** (color de fondo): color que se muestra por detrás del contenido y el espacio de relleno.
6. **Margin** (margen): separación opcional existente entre la caja y el resto de cajas adyacentes.

Dimensiones de las cajas

Anchura

Para los elementos de bloque y las imágenes, la propiedad width (anchura) permite establecer la anchura directamente mediante una medida.

Anchura = width

En CSS:

```
4    #lateral { width: 200px; }
```

En HTML:

```
11        <div id="lateral">…</div>
```

Si se utilizan unidades de medida, los valores indicados no pueden ser negativos. Si en vez de una unidad de medida se indica un porcentaje, la referencia de ese valor es la anchura del elemento que lo contiene. El valor heredado indica que la anchura del elemento se hereda de su elemento padre.

Si se establece la anchura de un elemento con la unidad de medida em, el valor indicado toma como referencia el tamaño de letra del propio elemento.

El valor auto es el valor por defecto de la anchura de todos los elementos. En este caso, el navegador determina la anchura de cada elemento teniendo en cuenta, entre otros, el tipo de elemento que se trata (de bloque o en línea), el sitio disponible en la pantalla del navegador y los contenidos de los elementos.

Altura

Al igual que sucede con ancho, la propiedad altura no admite valores negativos. Si se indica un porcentaje, se toma como referencia la altura del elemento padre. Si el elemento padre no tiene una altura definida explícitamente, se asigna el valor auto a la altura.

El valor heredado indica que la altura del elemento se hereda de su elemento padre. El valor automático, que es el que se utiliza si no se establece altura de forma explícita un valor a esta propiedad, indica que el navegador debe calcular automáticamente el elemento, teniendo en cuenta sus contenidos y el sitio disponible en la página.

Altura = height

En CSS:

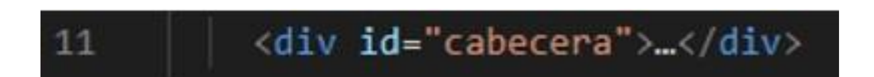
```
4    #cabecera { height: 60px; }
```







En HTML:



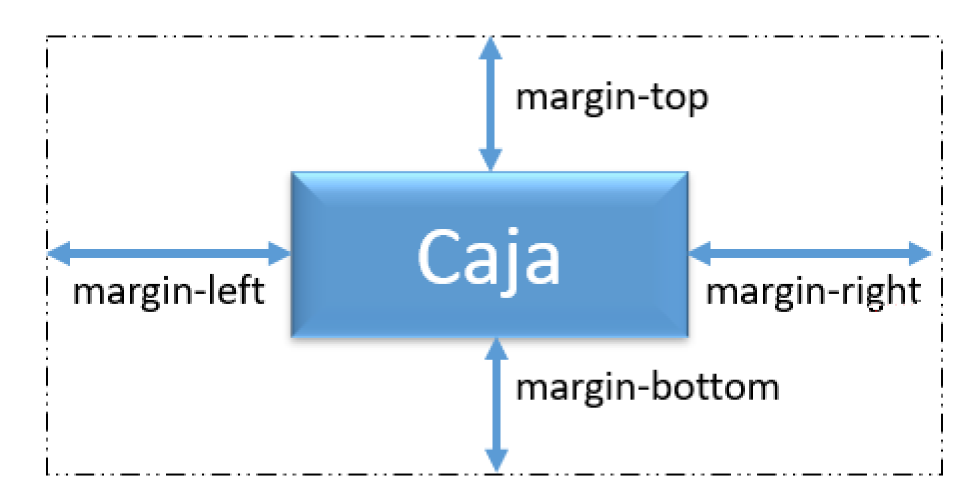
```
11    <div id="cabecera">…</div>
```

Margen

CSS define cuatro propiedades para controlar cada uno de los márgenes horizontales y verticales de un elemento.

1. margin-top => Margen superior
2. margin-right => Margen derecho
3. margin-bottom => Margen inferior
4. margin-left => Margen izquierdo

Cada una de las propiedades establece la separación entre el borde lateral de la caja y el resto de las cajas adyacentes:



Los márgenes verticales (margin-top y margin-bottom) sólo se pueden aplicar a los elementos de bloque y las imágenes, mientras que los márgenes laterales (margin-left y margin-right) se pueden aplicar a cualquier elemento.

Código CSS:

```css
div img {
  margin-top: 0.5em;
  margin-bottom: 0.5em;
  margin-left: 1em;
  margin-right: 0.5em;
}
```

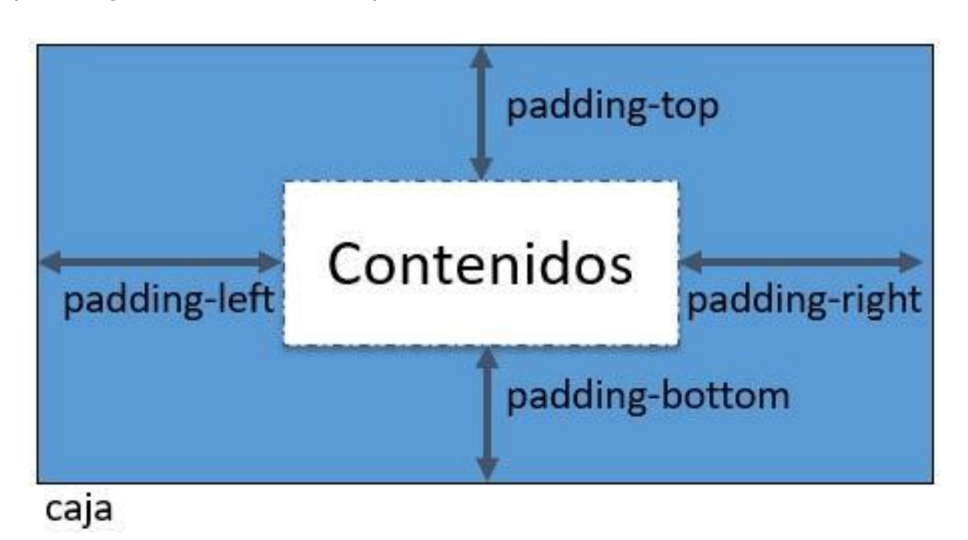
Alternativa:

```css
div img {
  margin: 0.5em 0.5em 0.5m 1em;
}
```

Relleno o padding

Relleno o padding. CSS define cuatro propiedades para controlar cada uno de los espacios de relleno horizontales y verticales de un elemento:

1. padding-top => Relleno superior
2. padding-right => Relleno derecho
3. padding-bottom => Relleno inferior
4. padding-left => Relleno izquierdo



Algunos ejemplos de padding:

```
body {
  padding: 2em;
} /* Todos los rellenos valen 2em */
body {
  padding: 1em 2em;
} /* Superior e inferior = 1em, Izquierdo y derecho = 2em */
body {
  padding: 1em 2em 3em;
} /* Superior = 1em, derecho = 2em, inferior = 3em, izquierdo = 2em */
body {
  padding: 1em 2em 3em 4em;
} /* Superior = 1em, derecho = 2em, inferior = 3em, izquierdo = 4em */
```

Unidades de medida

Antes de comenzar el ejercicio real, nos gustaría dar una breve idea sobre las unidades de medida CSS.

CSS admite una serie de medidas que incluyen unidades absolutas como pulgadas, centímetros, puntos, etc., así como medidas relativas como porcentajes y unidades em. Necesita estos valores al especificar varias medidas en sus reglas de estilo, por ejemplo, **border = "1px solid red"** .

Hemos enumerado todas las unidades de medida CSS junto con los ejemplos adecuados:

Unidad	Descripción	Ejemplo
%	Define una medida como un porcentaje relativo a otro valor, normalmente un elemento envolvente.	p {tamaño de fuente: 16pt; altura de línea: 125%;}
cm	Define una medida en centímetros.	div {margen inferior: 2 cm;}
en	Una medida relativa para la altura de una fuente en espacios em. Debido a que una unidad em es equivalente al tamaño de una fuente determinada, si asigna una fuente a 12 puntos, cada unidad "em" sería 12 puntos; por lo tanto, 2em serían 24 puntos.	p {espaciado entre letras: 7em;}
ex	Este valor define una medida relativa a la altura x de una fuente. La altura de x está determinada por la altura de la letra minúscula x de la fuente.	p {tamaño de fuente: 24pt; altura de línea: 3ex;}
en	Define una medida en pulgadas.	p {espaciado entre palabras: .15in;}
mm	Define una medida en milímetros.	p {espaciado entre palabras: 15 mm;}
pc	Define una medida en picas. Una pica equivale a 12 puntos; por tanto, hay 6 picas por pulgada.	p {tamaño de fuente: 20pc;}
por	Define una medida en puntos. Un punto se define como 1/72 de pulgada.	cuerpo {tamaño de fuente: 18pt;}
px	Define una medida en píxeles de la pantalla.	p {relleno: 25px;}

Colores

CSS usa valores de color para especificar un color. Por lo general, se utilizan para establecer un color para el primer plano de un elemento (es decir, su texto) o para el fondo del elemento. También se puede usar para afectar el color de los bordes y otros efectos decorativos.

Puedes especificar sus valores de color en varios formatos.

La siguiente tabla enumera todos los formatos posibles:

Formato	Sintaxis	Ejemplo
Código hexadecimal	#RRGGBB	p {color: # FF0000;}
Código hexadecimal corto	#RGB	p {color: # 6A7;}
RGB%	rgb (rrr%, ggg%, bbb%)	p {color: rgb (50%, 50%, 50%);}
Absoluto RGB	rgb (rrr, ggg, bbb)	p {color: rgb (0,0,255);}
palabra clave	aguamarina, negro, etc.	p {color: verde azulado;}

Fuentes

Veamos cómo configurar las fuentes de un contenido, disponible en un elemento HTML. Puedes establecer las siguientes propiedades de fuente de un elemento:

- La propiedad **font-family** se utiliza para cambiar la cara de una fuente.
- La propiedad de **font-style** se usa para hacer una fuente en cursiva u oblicua.
- La propiedad **font-variant** se utiliza para crear un efecto de versalitas.
- La propiedad de **font-weight** se utiliza para aumentar o disminuir la negrita o la luz de una fuente.
- La propiedad de **font-size** se utiliza para aumentar o disminuir el tamaño de una fuente.
- La propiedad de **font** se utiliza como forma abreviada para especificar otras propiedades de fuente.

Bootstrap

Introducción

El siguiente curso que estás por ver, trata sobre un framework de interfaz de usuario o UI según sus siglas en inglés. Si ya has programado anteriormente sitios web seguramente lo conoces. Bootstrap se ha convertido en unos de los frameworks que utilizan HTML, CSS y JS más popular y a su vez más utilizado por los desarrolladores. Su potente sistema de grilla, su extensa y clara documentación, su gran capacidad de adaptación a los distintos tamaños de dispositivos, su variada colección de componentes y la extensa cantidad de templates desarrollados con bootstrap son algunas de las razones, por lo cual hoy, es tan elegido.

Objetivos

El objetivo de este tutorial es brindarles una guía de inicio al framework que les facilite su pronta incorporación a sus proyectos. En la misma trataremos los temas más importantes del mismo tales como instalación, reglas para su uso, características principales entre otros. Al finalizar la lectura de este tutorial ya

debemos estar en condiciones de hacer uso de bootstrap en nuestras aplicaciones y tener la capacidad para comenzar a profundizar sobre otros aspectos que se pueden encontrar en su documentación oficial.

Historia

Bootstrap, originalmente llamado Blueprint de Twitter, fue desarrollado por Mark Otto y Jacob Thornton de Twitter, como un marco de trabajo (framework) para fomentar la consistencia entre las herramientas internas.

Antes de Bootstrap, se usaron varias bibliotecas para el desarrollo de interfaces de usuario, lo que generó inconsistencias y una gran carga de trabajo en su mantenimiento.

El primer desarrollo en condiciones reales ocurrió durante la primera "Semana de Hackeo" (Hackweek) de Twitter. Mark Otto mostró a algunos colegas cómo acelerar el desarrollo de sus proyectos con la ayuda de la herramienta de trabajo. Como resultado, decenas de temas se han introducido en el marco de trabajo.

En agosto del 2011, Twitter liberó Bootstrap como código abierto. En febrero del 2012, se convirtió en el proyecto de desarrollo más popular de GitHub.

Versiones

Bootstrap utiliza un versionado semántico incremental “Principal.Menor.Parche”. La versión “Principal” se utiliza para un cambio importante e incompatible con la versión anterior, actualmente se encuentra en la versión v5.0.1. La versión “Menor” agrega funcionalidad de una manera compatible con la misma versión principal. Por ejemplo, la v3.4.2 significa que es compatible con la versión 3. Finalmente, el cambio “Parche” se utiliza cuando realiza correcciones de errores dentro de la misma versión menor.

Instalación

En el siguiente apartado mostraremos cómo incorporar fácil y rápidamente el framework a un proyecto, describiendo tres de las principales formas de instalación. El funcionamiento de Bootstrap requiere tanto de su propia librería de javascript como la de Popper (https://popper.js.org/), esta característica hace que se pueda instalar tanto atado (bundle en inglés) o por separado.

Es importante recordar que la referencia a los archivos CSS se debe incluir dentro de la etiqueta <head>...</head> mediante la etiqueta <link>, un ejemplo sería <head>... <link href="dist/bootstrap/css/bootstrap.min.css" rel="stylesheet"></head>. Para el caso de los archivos JS (javascript) preferentemente se deben incluir al final de la etiqueta <body> mediante la etiqueta







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

<script>, un ejemplo de que incluye la librería de bootstrap y popper sería <body> … <script src="dist/bootstrap/js/bootstrap.bundle.min.js"></script> </body>.

1. CSS y Javascript compilados y/o minimizados

Esta forma de instalación requiere que los archivos de Bootstrap sean descargados previamente de forma local. Por ejemplo, la última versión disponible al momento de la edición de este documento, se encuentra en el siguiente enlace https://github.com/twbs/bootstrap/releases/download/v5.0.1/bootstrap-5.0.1-dist.zip.

Luego se incorporan a la estructura del proyecto, para finalmente poder referenciar los archivos del framework . Cumpliendo algunos estándares del desarrollo web, para el ejemplo se copian los archivos dentro de la carpeta “dist” (distribución), pero esta puede tener cualquier otro nombre.

Ejemplo 1: Instalación atada, compilada y minimizada

```
<!doctype html>

<html lang="en">

  <head>

    <!-- Required meta tags -->

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width,initial-scale=1">

    <!-- Bootstrap CSS -->

    <link href="dist/bootstrap/css/bootstrap.min.css" rel="stylesheet">

    <title>Instalación atada, compilada y minimizada</title>

  </head>

  <body>

    <h1>Atado, compilado y minimizado</h1>

    <!-- Bootstrap JS -->
```

```
    <script src="dist/bootstrap/js/bootstrap.bundle.min.js"></script>
  </body>
</html>
```

Ejemplo 2: Instalación atada y compilada

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <!-- Bootstrap CSS -->
    <link href="dist/bootstrap/css/bootstrap.css" rel="stylesheet">
    <title>Instalación atada y compilada</title>
  </head>
  <body>
    <h1>Atado y compilado</h1>
    <!-- Bootstrap JS -->
    <script src="dist/bootstrap/js/bootstrap.bundle.js"></script>
  </body>
</html>
```







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

Ejemplo 3: Instalación separada, compilada y minimizada

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <!-- Bootstrap CSS -->
    <link href="dist/bootstrap/css/bootstrap.min.css" rel="stylesheet">
    <title>Instalación separada, compilada y minimizada</title>
  </head>
  <body>
    <h1>Separado, compilado y minimizado</h1>
    <!-- Bootstrap JS -->
    <script src="dist/bootstrap/js/bootstrap.min.js"></script>
    <script src="dist/bootstrap/js/bootstrap.esm.min.js"></script>
  </body>
</html>
```

Ejemplo 4: Instalación separada, compilada y minimizada

```
<!doctype html>
<html lang="en">
  <head>
```

```
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <!-- Bootstrap CSS -->
    <link href="dist/bootstrap/css/bootstrap.css" rel="stylesheet">
    <title>Instalación separada y compilada</title>
  </head>
  <body>
    <h1>Separado y compilado</h1>
    <!-- Bootstrap JS -->
    <script src="dist/bootstrap/js/bootstrap.js"></script>
    <script src="dist/bootstrap/js/bootstrap.esm.js"></script>
  </body>
</html>
```

CDN a través de jsDelivr

La instalación a través de una Red de Distribución de Contenido (*content delivery network o CDN según siglas en inglés*) reemplaza la necesidad de descargar el contenido del framework y de referenciar sus archivos en forma local. La Red de Distribución de Contenido propuesta por Bootstrap de manera oficial para su versión actual (5.0.1) es jsDelivr.

```
<!-- Bootstrap CSS -->

<head>...<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-+0n0xVW2eSR5OomGNYDnhzAbDsOXxcvSN1TPprVMTNDbiYZCxYbOOl7+AMvyTG2x" crossorigin="anonymous"></head>
```

<!-- Bootstrap JS -->

<body>...<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/js/bootstrap.bundle.min.js" integrity="sha384-gtEjrD/SeCtmISkJkNUaaKMoLD0//ElJ19smozuHV6z3Iehds+3Ulb9Bn9Plx0x4" crossorigin="anonymous"></script></body>

Diseño

El uso correcto del framework hace que los desarrolladores tengan que obedecer cierto estándar en el cumplimiento de algunas reglas, maquetando las páginas web de una forma muy rápida, permitiendo abstraerse de las configuraciones propias de css.

Todas los elementos deben estar contenidos dentro de un contenedor con la clase ***.container*** o ***.container-fluid***. Ejemplo <div class="container">...</>.



Fuente de la imagen: https://www.w3schools.com/bootstrap4/bootstrap_containers.asp

Contenedores

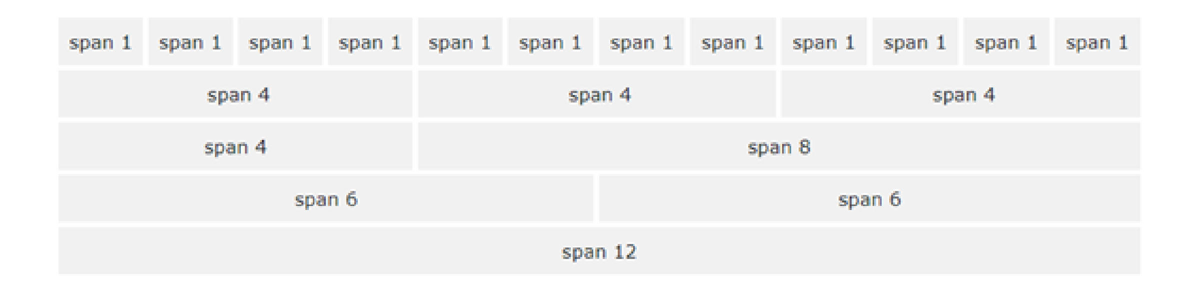
Tal como se indicó al inicio de esta sección, los contenedores son el espacio de construcción fundamental que incluye Bootstrap, y es donde se ajustan y alinean su contenido dentro de un dispositivo o ventana gráfica determinada.

	Extra pequeño <576px	**Pequeño ≥576px**	**Medio ≥768px**	**Grande ≥992px**	**X-grande ≥1200px**	**XX-Large ≥1400px**
.container	100%	540 px	720px	960 px	1140px	1320px
.container-sm	100%	540 px	720px	960 px	1140px	1320px
.container-md	100%	100%	720px	960 px	1140px	1320px
.container-lg	100%	100%	100%	960 px	1140px	1320px

.container-xl	100%	100%	100%	100%	1140px	1320px
.container-xxl	100%	100%	100%	100%	100%	1320px
.container-fluid	100%	100%	100%	100%	100%	100%

Sistema de grillas

El sistema Grid permite dividir la página web en 12 columnas permitiendo adaptar la disposición de los elementos como se muestra abajo fácilmente.



Antes de continuar, es importante mencionar algunas reglas a la hora de trabajar con el sistema de grillas de Bootstrap:

- Las filas deben estar dentro de un contenedor (.container o .container-fluid).
- Usar filas para crear grupos de columnas. No a la inversa. Esto es posible agregando la clase .row al contenedor horizontal y la clase col al contenedor columna. Ejemplo <div class=”row”><div class =”col”>Contenido</div></div>.
- El contenido debe estar dentro de las columnas.
- Los únicos elementos anidados dentro de una fila (row) deben ser columnas (col).
- Las columnas se deben crear especificando el número de columnas disponibles (12).

A continuación, se muestra la estructura básica del sistema de grillas que propone Bootstrap:

```
<!-- Control the column width, and how they should appear on different devices -->
<div class="row">
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
</div>
<div class="row">
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
</div>

<!-- Or let Bootstrap automatically handle the layout -->
<div class="row">
  <div class="col"></div>
  <div class="col"></div>
  <div class="col"></div>
</div>
```

Columnas de Ancho Automático

Para definir columnas de ancho automático simplemente debemos utilizar la clase “col” como se muestra en el siguiente ejemplo:

Column	Column	Column

```
<div class="container">
  <div class="row">
    <div class="col"> Column </div>
    <div class="col"> Column </div>
    <div class="col"> Column </div>
  </div>
</div>
```

Columnas de Ancho Fijo

Las columnas de ancho automático distribuyen el ancho de la pantalla equitativamente para cada columna pero ¿y si deseamos establecer un ancho fijo?

En este caso, simplemente debemos especificar el número de columnas como sigue:

.col-4	.col-8

```
<div class="container">

  <div class="row">

    <div class="col-4"> Column </div>

    <div class="col-8"> Column </div>

  </div>

</div>
```

Las columnas se deben crear especificando de manera explícita teniendo en cuenta el número de columnas disponibles, cuya cantidad total es 12.

La combinación del sistema de grillas con los puntos de interrupción es la unión perfecta para colocar nuestro contenido en la web según cada criterio de diseño y con la posibilidad de adaptarlo y ordenarlo al tamaño de pantalla donde se esté mostrando.

Nota: El sistema de grilla utiliza una cuadrícula flexbox para su diseño.

	xs **<576px**	**sm** **≥576px**	**md** **≥768px**	**lg** **≥992px**	**xl** **≥1200px**	**xxl** **≥1400px**
Envase max-width	Ninguno (automático)	540 px	720px	960 px	1140px	1320px
Prefijo de clase	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-	.col-xxl-
# de columnas	12					

Puntos de interrupción

El framework para lograr un diseño ajustable o responsive incluye seis puntos de interrupción o de ruptura, predeterminados. El uso eficiente de estos puntos, como diseñadores o maquetadores web, nos habilita una herramienta para combinar en conjunto al sistema de grillas que propone bootstrap para generar contenidos que se ajusten correctamente a la pantalla del dispositivo donde se esté mostrando.

Por esta razón debemos conocer en detalle cuáles son los infijos de clases y el rango de píxeles de pantallas en los que son identificados.

Nota: Es importante recordar que el concepto responsive hace alusión a un diseño que sea accesible y adaptable a todos los dispositivos.

Breakpoint	Infijo de clase	Dimensiones
X-pequeño	*Ninguno*	<576 px
Pequeña	sm	≥576px
Medio	md	≥768px
Grande	lg	≥992px
Extra grande	xl	≥1200px
Extra extra grande	xxl	≥1400px

https://getbootstrap.com/docs/5.0/layout/breakpoints/

Columnas Responsive

De acuerdo a lo expresado arriba, podemos utilizar los puntos de interrupción para lograr un diseño ajustable como sigue:

.col-sm-4	.col-sm-8

```
<div class="container">

  <div class="row">

    <div class="col-sm-4"> .col-sm-4 </div>

    <div class="col-sm-8"> .col-sm-8 </div>

  </div>

 </div>
```

Columnas de Ancho Mixto

Podemos combinar las columnas de ancho automático y fijo como sigue:

.col	.col-6	.col

```
<div class="container">
```

```
  <div class="row">

    <div class="col"> .col</div>

    <div class="col-6"> .col-6 </div>

<div class="col"> .col</div>
```

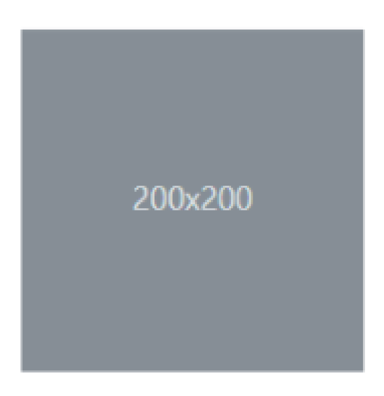
Imagenes

El framework, para el caso de las imágenes, nos propone una serie de clases con el objetivo de hacerlas adaptables y/o que nunca sean más grandes que las columnas que la contienen.

Imagen Responsive

```
<img src="..." class ="img-fluid" alt="...">
```

Ejemplo: Imágenes en miniaturas (200 x 200) con un borde de 1px




```
<img src="..." class= "img-thumbnail" alt="...">
```

Ejemplo: Alinear imágenes (1 al inicio de la fila y 2 al final de la fila)

200x60 200x60

```
<img src="..." class ="rounded float-start" alt="...">

<img src="..." class= "rounded float-end" alt="...">
```

Tablas

El uso de tablas en bootstrap es opcional, ya que no es un elemento que se herede del framework. Dado a su uso popular por los programadores, se propone un grupo de clases para trabajar con los estilos de una tabla.

Ejemplo:

Nro	Nombre	Apellido
1	Maximiliano	Guerra
2	Erik	Thomas

```
<table class="table">

  <thead>

    <tr>

      <th scope="col">Nro</th>

      <th scope="col">Nombre</th>

      <th scope="col">Apellido/th>

    </tr>

  </thead>

  <tbody>

    <tr>

      <th scope="row">1</th>

      <td>Maximiliano</td>

      <td>Guerra</td>

    </tr>

    <tr>
```

```
        <th scope="row">2</th>

        <td>Erik</td>

        <td>Thomas</td>

      </tr>

    </tbody>

  </table>
```

También se puede hacer uso de las clases contextuales para dar color a las tablas, de la misma forma que a cualquier otro elemento.

Class	Heading	Heading
Default	Cell	Cell
Primary	Cell	Cell
Secondary	Cell	Cell
Success	Cell	Cell
Danger	Cell	Cell
Warning	Cell	Cell
Info	Cell	Cell
Light	Cell	Cell
Dark	Cell	Cell

```
<!-- On tables -->

<table class="table-primary">...</table>

<table class="table-secondary">...</table>

<table class="table-success">...</table>

<table class="table-danger">...</table>

<table class="table-warning">...</table>
```

```
<table class="table-info">...</table>

<table class="table-light">...</table>

<table class="table-dark">...</table>

<!-- On rows -->

<tr class="table-primary">...</tr>

<tr class="table-secondary">...</tr>

<tr class="table-success">...</tr>

<tr class="table-danger">...</tr>

<tr class="table-warning">...</tr>

<tr class="table-info">...</tr>

<tr class="table-light">...</tr>

<tr class="table-dark">...</tr>


 <!-- On cells (`td` or `th`) -->

<tr>

  <td class="table-primary">...</td>

  <td class="table-secondary">...</td>

  <td class="table-success">...</td>

  <td class="table-danger">...</td>

  <td class="table-warning">...</td>

  <td class="table-info">...</td>

  <td class="table-light">...</td>

  <td class="table-dark">...</td>
```

```
</tr>
```

Figuras

Al usar imágenes con texto relacionado se recomienda el uso de la etiqueta <*figure*>. La etiqueta <*figcaption*> contiene el texto de la imagen, pudiendo este ser alineado de igual manera que los elementos de texto.

Ejemplo:

Figura 1.2

Figura 1.2

```
<figure class="figure">

  <img src="..." class="figure-img img-fluid rounded" alt="...">

  <figcaption class="figure-caption">Figura 1.2</figcaption>

</figure>
```

Formularios

Al igual que las imágenes, bootstrap provee un grupo de clases para la presentación de los controles dentro de un formulario.

Ejemplo:

Correo

Debe ser un correo válido.

Password

☐ Recordarme

Enviar

```
<form>
```








```
    <div class="mb-3">
      <label for="idE" class="form-label">Correo</label>
      <input type="email" class="form-control" id="idE" aria-describedby ="idEH">
      <div id="idEH" class="form-text">Debe ser un correo válido.</div>
    </div>
    <div class="mb-3">
      <label for="exampleInputPassword1" class="form-label">Password</label>
      <input type="password" class="form-control" id="exampleInputPassword1">
    </div>
    <div class="mb-3 form-check">
      <input type="checkbox" class="form-check-input" id="exampleCheck1">
      <label class="form-check-label" for="exampleCheck1">Recordarme</label>
    </div>
    <button type="submit" class="btn btn-primary">Enviar</button>
</form>
```

Controles

Los controles dentro de un formulario pueden ser personalizados e incluso utilizados de forma adaptativa, gracias a las clases provistas por bootstrap.

Dimensionamiento:

- .form-control-sm (tamaño pequeño)
- .form-control (tamaño normal)
- .form-control-lg (tamaño grande)

Habilitado:

- Atributo **disabled** (Control deshabilitado)
- Atributo **readonly** (Control sólo lectura)
- Atributos **disabled readonly** (Control deshabilitado, sólo lectura)

Layout

“Cada grupo de campos de formulario debe residir en un elemento “<form>”. “Bootstrap no proporciona un estilo predeterminado para el <form> elemento, pero hay algunas funciones de navegador poderosas que se proporcionan de forma predeterminada.”

● *“<button>*s dentro de un *<form>* valor predeterminado de type="submit", así que esfuérzate por ser específico y siempre incluye un archivo type.

● Puedes deshabilitar todos los elementos del formulario dentro de un formulario con el disabled atributo en *<form>*.

Ya que Bootstrap se emplea display: block y width: 100% para casi todos los controles del formulario, los formularios se mostrarán por defecto de manera vertical. Se pueden usar clases adicionales para variar este diseño en función de cada formulario.”

Componentes

Alertas

Este componente es muy utilizado a la hora de mostrar mensajes al usuario de forma flexible y de una manera más amigable. Las alertas son consecuentes con los estilos de colores que propone el framework.

Ejemplo:

A simple primary alert—check it out!

A simple secondary alert—check it out!

A simple success alert—check it out!

A simple danger alert—check it out!

A simple warning alert—check it out!

A simple info alert—check it out!

A simple light alert—check it out!

A simple dark alert—check it out!

```
<div class="alert alert-primary" role="alert">
    A simple primary alert—check it out!
</div>
<div class="alert alert-secondary" role="alert">
  A simple secondary alert—check it out!
</div>
<div class="alert alert-success" role="alert">
  A simple success alert—check it out!
</div>
<div class="alert alert-danger" role="alert">
  A simple danger alert—check it out!
```








```
</div>

<div class="alert alert-warning" role="alert">

  A simple warning alert—check it out!

</div>

<div class="alert alert-info" role="alert">

  A simple info alert—check it out!

</div>

<div class="alert alert-light" role="alert">

  A simple light alert—check it out!

</div>

<div class="alert alert-dark" role="alert">

  A simple dark alert—check it out!

</div>
```

Botones

Al igual que el resto de los controles, los botones incluyen el estilo predefinido.

Ejemplo:

```
<button type="button" class="btn btn-primary">Primary</button>

<button type="button" class="btn btn-secondary">Secondary</button>

<button type="button" class="btn btn-success">Success</button>

<button type="button" class="btn btn-danger">Danger</button>

<button type="button" class="btn btn-warning">Warning</button>

<button type="button" class="btn btn-info">Info</button>

<button type="button" class="btn btn-light">Light</button>

<button type="button" class="btn btn-dark">Dark</button>
```

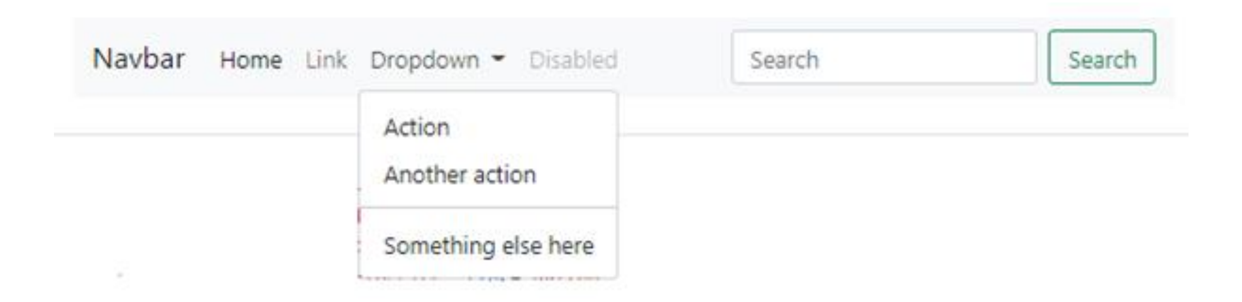
<button type="button" class="btn btn-link">Link</button>



Barra de navegación

Por último, abordaremos el tema de la barra de navegación, elemento muy utilizado en el desarrollo de aplicaciones web. A continuación, citamos las principales reglas para el uso de la barra de navegación.

- Las barras de navegación requieren comenzar con la clase .navbar
- Las barras de navegación y su contenido son fluidos por defecto.
- Las barras de navegación responden de forma predeterminada, pero se pueden modificar fácilmente para cambiar su comportamiento.
- Garantice la accesibilidad mediante el uso de un elemento <nav> o, si usa un elemento más genérico como <div>, agregue a role="navigation" a cada barra de navegación.



Introducción a JavaScript

Según Mozilla (MDN 2020), JavaScript, o simplemente JS, es un lenguaje de scripting multiplataforma y orientado a objetos. Es un lenguaje pequeño y liviano. Dentro de un ambiente de host, JavaScript puede conectarse a los objetos de su ambiente y proporcionar control programático sobre ellos. Se trata de un lenguaje de programación tipo script, basado en objetos y guiado por eventos, diseñado específicamente para el desarrollo de aplicaciones cliente-servidor dentro

del ámbito de Internet.

Los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.
JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos.

Objetivos

Introduciremos el concepto de programación estructurada y el lenguaje de programación interpretado JavaScript. Nos enfocaremos en JavaScript del lado del cliente, lo cual proporciona además una serie de objetos para controlar un navegador y su modelo de objetos (o DOM, por las iniciales en inglés de Document Object Model). Por ejemplo, las extensiones del lado del cliente permiten que una aplicación coloque elementos en un formulario HTML y responda a eventos del usuario, tales como clics, ingreso de datos al formulario, navegación de páginas, etc.

Un poco de historia

¿Por qué se llama Javascript ?, según Crockford (2020), el prefijo de Java sugiere que Javascript está de algún modo relacionado con Java, que es un subconjunto de instrucciones de Java o que es una versión menos potente. (...) "*Parece que el nombre fue intencionalmente seleccionado para crear confusión y de esa confusión surge la mala interpretación*" (...).

Javascript es un lenguaje diferente a Java. Si bien Javascript tiene una sintaxis similar a Java, no significa que tenga algo que ver con Java. Java tiene una sintaxis similar al lenguaje C y esto no quiere decir que Java sea C. El lenguaje C es otro lenguaje de programación.

Javascript fue creado por Brendan Eich en 1995 para la empresa (y navegador popular en su época) Netscape. "*Como Java era un lenguaje nuevo en esa era y con mucha aceptación, se optó por utilizar parte del nombre para ganar adeptos*". (Brendan Eich, 2016).

Por último y para cerrar este apartado, vamos a decir que Javascript está estandarizado en Ecma International (asociación europea para la creación de estándares para la comunicación y la información) con el fin de ofrecer un lenguaje de programación estandarizado e internacional basado en Javascript y llamado ECMAScript para que todos los navegadores puedan realizar un intérprete que acepte la misma sintaxis de Javascript.

Sintaxis de JavaScript

Sintaxis

A continuación vamos a ver la sintaxis del lenguaje JS (JavaScript) para familiarizarnos con la forma en que se codifica. Antes de comenzar con apartados más específicos veremos cuestiones generales sobre el lenguaje.

Javascript es un lenguaje que distingue entre mayúsculas y minúsculas. Por ejemplo, identificador (o nombre) “OnClickEvent” es distinto a “onclickevent”. Esta característica hace que prestemos mucha atención a la forma correcta de llamar usar y/o llamar a los identificadores. Las instrucciones son llamadas sentencias y son separadas por punto y coma (;) al final de la misma, si bien no es necesario en todos los casos es una buena práctica hacerlo.

Existen dos formas de realizar comentarios: MultiLine o SingleLine. Para el caso de multilínea se utilizan los caracteres “/ *” y “* /” por ejemplo:

/ * Esto es un comentario

multilínea * /

En el caso de un comentario de línea solamente es necesario usar los caracteres “//” al inicio de la línea a comentar, por ejemplo:

// Esto es un comentario de una sola línea

Por último, debemos tener en cuenta que como cualquier otro lenguaje de programación, JS también usa palabras reservadas (palabras clave) que no podrán utilizarse por los programadores para usarlas como identificadores. Algunos ejemplo de ellas son: await, break, case, catch, class, const, continue, debugger, default, delete, do, else, enum, export, extensions, false, finalmente, for, function, if, import, in, instanceof, new, null, return, super, switch, this, throw, true, try, typeof, var, void, while, with, yield, etc.

Tipos de datos en JavaScript

El último estándar ECMAScript define nueve tipos:

- Seis tipos de datos primitivos, controlados por el operador typeof
    1. **Indefinido**: tipo de instancia === "indefinido"
    2. **Booleano**: tipo de instancia === "booleano"
    3. **Número**: tipo de instancia === "número"
    4. **Cadena**: tipo de instancia === "cadena"
    5. **BigInt**: tipo de instancia === "bigint"
    6. **Símbolo**: tipo de instancia === "símbolo"

- 7. **Nulo:** tipo de instancia === "objeto". Tipo primitivo especial que tiene un uso adicional para su valor: si el objeto no se hereda, se muestra null.

- 8. **Objeto:** tipo de instancia === "objeto". Tipo estructural especial que no es de datos, pero para cualquier instancia de objeto construido que también se utiliza como estructuras de datos: new Object, new Array, new Map, new Set, new WeakMap, new WeakSet, new Date y casi todo lo hecho con la palabra clave nuevo

- 9. **Función:** una estructura sin datos, aunque también respondió al operador typeof: typeof instance === "function". Esta simplemente es una forma abreviada para funciones, aunque cada constructor de funciones se deriva del constructor Object.

Nota: El único propósito valioso del uso del operador typeof es verificar el tipo de dato. Si deseamos verificar cualquier Tipo Estructural derivado de Object, no tiene sentido usar typeof para eso, ya que siempre recibiremos "object". La forma correcta de comprobar qué tipo de Objeto estamos usando es la palabra clave instanceof.

Variables y Ámbitos en JavaScript

El concepto de variable nos va a ser útil durante todo el proceso de programación en los distintos lenguajes. Javascript implementa el mismo concepto: Las variables son un espacio designado en la memoria para almacenar un dato. Se le asocia un nombre que la identifica y un tipo de datos.

Ejemplo de declaración y asignación de valor:

var mi_dato_numerico = 3;

var mi_dato_texto = "hola";

Declaraciones:

En JS existen tres tipos de declaraciones de variables:

1. Usando var: declara una variable, inicializándola opcionalmente a un valor. Si bien esta ha sido exclusivamente la forma de declarar variables durante mucho tiempo, sufre de unos efectos no deseados en el alcance (scope) de la declaración. En otras palabras, se aconseja no continuar utilizándola ya que el

nuevo estándar de javascript permite otras formas de declaración que corrigen este problema.

El alcance o bloque de ámbito es el lugar en donde es visible esa variable. Por ejemplo, si se declara una variable dentro de una función, esta variable mantiene su valor y es visible sólo dentro de una función.

2. Usando let: declara una variable local en un bloque de ámbito (scope), inicializándola opcionalmente a un valor.

3. Usando const: declara una constante de sólo lectura en un bloque de ámbito. Una constante funciona como una variable pero no cambia su valor, es decir, representa un lugar de almacenamiento de tipos de datos en la memoria pero una vez asignado el valor inicial este no puede ser modificado. La sintaxis de la definición del identificador es la misma que la de las variables.

Las variables se usan como nombres simbólicos para valores en tu aplicación. Los nombres de las variables, llamados identificadores, se rigen por ciertas reglas. Un identificador en JavaScript tiene que empezar con una letra, un guion bajo (_) o un símbolo de dólar ($). Los valores subsiguientes pueden ser números. Debido a que JavaScript diferencia entre mayúsculas y minúsculas, las letras incluyen tanto desde la "A" hasta la "Z" (mayúsculas) como de la "a" hasta la "z". Por ejemplo:

var _numero = 100;

var numero = 100;

let $numero = 100;

const numeroPar = 100;

const PI = 3.14;

Ámbito

A partir de la versión ES6 / ECMAScript 2015, Javascript habilita tres ámbitos para la declaración de una variable:

1. Ámbito Global: Cuando se declara una variable fuera de una función, se le denomina variable global, porque está disponible para cualquier otro código en el documento actual.

2. Ámbito Bloque: Este nuevo ámbito es utilizado para las variables declaradas con la palabra reservada let / const dentro de un bloque de código delimitados por llaves {}, esto implica que no pueden ser accesibles fuera de este bloque.







3. Ámbito: Función: Cuando se declara una variable dentro de una función, se le denomina variable local, porque está disponible sólo dentro de esa función donde fue declarada.

Operadores en JavaScript

Los operadores en los lenguajes de programación son símbolos que indican cómo se deben manipular los operadores. Los operadores en conjunto con los operandos forman una expresión, una fórmula que define el cálculo de un valor. Los operandos pueden ser constantes, variables o llamadas a funciones.

Operadores matemáticos

Operador		Descripción	Ejemplo
+	Suma	Suma dos números	let suma = 5 + 7; // = 13
-	Resta	Resta dos números	let resta = 7-2; // = 5
*	Multiplicación	Multiplica dos números	let mul = 5 * 7; // = 35
/	División	Dividir dos números	let div = 15/3; // = 5
%	Módulo	Devuelve el resto de dividir de dos números	let mod = 17% 4; // = 1
++	Incremento	Suma 1 valor al contenido de una variable	let i = 1; i ++; // = 2
–	Decremento	Resta un valor al contenido de una variable	let i = 2; i --; // = 1

Operadores lógicos

Operador		Descripción
==	Igualdad	Devuelve verdadero (true) si ambos operandos son iguales.
! =	Desigualdad	Devuelve verdadero (true) si ambos operandos no son iguales.
> =	Mayor o igual que	Devuelve verdadero (true) si el operando de la izquierda es mayor o igual que el operando de la derecha.
>	Mayor	Devuelve verdadero (true) si el operando de la izquierda es mayor que el operando de la derecha.
<	Menor	Devuelve verdadero (true) si el operando de la izquierda es menor que el operando de la derecha.





= <	Menor o igual que	Devuelve verdadero (true) si el operando de la izquierda es menor o igual que el operando de la derecha

Operador condicional (ternario)

El operador condicional es el único operador de JavaScript que necesita tres operadores. El operador asigna uno de dos valores basado en una condición.

La sintaxis de este operador es: condición? valor1: valor2. Si la condición es verdadera, el operador tomará el valor 1, de lo contrario tomará el valor 2. Se puede usar el operador condicional en cualquier lugar que use un operador estándar. Ejemplo: esta sentencia asigna el valor adulto a la variable estado si la edad es mayor o igual a 18, de lo contrario le asigna el valor menor:

```
var estado = (edad>= 18)? "adulto": "menor";
```

Funciones y estructuras en JavaScript

Switch

Una sentencia **switch** permite a un programa evaluar una expresión e intentar igualar el valor de dicha expresión a una etiqueta de caso (case). Si se encuentra una coincidencia, el programa ejecuta la sentencia asociada. Una sentencia **switch** se describe como se muestra a continuación:

```
switch(expresión) {

   case etiqueta_1:
      sentencias_1
      break;
   case etiqueta_2:
      sentencias_2
      break;
   ...
   default:
      sentencias_por_defecto
```

break;

}

Primero busca una cláusula case con una etiqueta que coincide con el valor de la expresión y entonces, transfiere el control a esa cláusula, ejecutando las sentencias asociadas a ella. Si no se encuentran etiquetas coincidentes, busca la cláusula opcional predeterminada y transfiere el control a esa cláusula ejecutando las sentencias asociadas. Si no se encuentra la cláusula predeterminada, el programa continúa su ejecución por la siguiente sentencia al final del switch. Por convención la cláusula por defecto es la última cláusula aunque no es necesario que sea así.

La sentencia opcional break asociada con cada cláusula case asegura que el programa finaliza la sentencia switch una vez que la sentencia asociada a la etiqueta coincidente es ejecutada y continúa la ejecución por las sentencias siguientes a la sentencia switch. Si se omite la sentencia break, el programa continúa su ejecución por la siguiente sentencia que haya en la sentencia switch.

Funciones

Las funciones son uno de los bloques de construcción fundamentales en JavaScript. Una función en JavaScript es similar a un procedimiento - un conjunto de instrucciones que realiza una tarea o calcula un valor, pero para que un procedimiento califique como función, debe tomar alguna entrada y devolver una salida donde hay alguna relación obvia entre la entrada y la salida. Para usar una función, debes definirla en algún lugar del ámbito desde el que deseas llamarla.

La declaración de una función consiste en:

1. Un nombre
2. Una lista de parámetros o argumentos encerrados entre paréntesis.
3. Conjunto de sentencias JavaScript encerradas entre llaves.

función nombre (parámetro1, parámetro2)

{

// código ha ser ejecutado;

}

En resumen, es un conjunto de instrucciones o sentencias que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente. Realizan varias

operaciones invocando un nombre. Esto permite simplificar el código pudiendo crear nuestras propias funciones y usarlas cuando sea necesario.

Estructuras repetitivas

Las estructuras repetitivas o cíclicas ejecuta varias veces un conjunto de instrucciones. A estas repeticiones se las conoce con el nombre de ciclos o bucles.

Los lenguajes de programación son muy útiles para completar rápidamente tareas repetitivas, desde múltiples cálculos básicos hasta cualquier otra situación en la que tengas un montón de elementos de trabajo similares que completar. Aquí vamos a ver las estructuras de bucles disponibles en JavaScript que pueden manejar tales necesidades:

Estructuras repetitivas en JavaScript:

- **FOR**
- **WHILE**
- **DO - WHILE**

Sentencia for (for)

Un bucle for (para) se repite hasta que la condición especificada se evalúa como falsa.

```
for ([expresion-inicial]; [condicion]; [expresion-final]) {

   // código ha ser ejecutado;

}
```

Cuando un bucle for se ejecuta, ocurre lo siguiente: la [expresión-inicial], si existe, se ejecuta. Esta expresión habitualmente inicializa uno o más contadores del bucle, pero la sintaxis permite una expresión con cualquier grado de complejidad. Esta expresión puede también declarar variables. Se evalúa la expresión [condición]. Si el valor de condición es true, se ejecuta la sentencia del bucle. Si el valor de condición es falso, el bucle para, finaliza. Si la expresión condición es omitida, la condición es asumida como verdadera. Se ejecuta la expresión [expresión-final], si hay una, y el control vuelve a evaluar la expresión [condición].

El ejemplo a continuación muestra un ciclo que se ejecuta 10 veces e imprime por consola la expresión “Número: X” donde x va del valor 0 al 9.

```
for (var i = 0; 1 <10; i ++) {
   console.log ("Número:" + i);
}
```

Sentencia while (mientras)

Una sentencia while ejecuta sus sentencias mientras la condición sea como verdadera. Una sentencia while tiene el siguiente aspecto:

```
while ([condicion]) {
   // código ha ser ejecutado;
}
```

Si la condición cambia a falsa, la sentencia dentro del bucle deja de ejecutarse y el control pasa a la sentencia inmediatamente después del bucle. La condición se evalúa antes de que la sentencia contenida en el bucle sea ejecutada. Si la condición devuelve verdadero, la sentencia se ejecuta y la condición se comprueba de nuevo. Si la condición es como falso, se detiene la ejecución y el control pasa a la sentencia siguiente al while.

Sentencia do-while (hacer - mientras)

Se utiliza para repetir instrucciones un número indefinido de veces, hasta que se cumpla una condición. A diferencia de la estructura mientras (while), la estructura hasta (do while) se ejecutará al menos una vez. Ejemplo:

```
let resultado = '';

let i = 0;

do {

  i = i + 1;

  resultado = resultado + i;

} while (i <5);

console.log (resultado);
```

// resultado esperado: "12345"

Sentencia continue (continuar)

La sentencia continue puede usar para reiniciar una sentencia for, while o do-while, continue termina la iteración en curso del código y continúa la ejecución del bucle con la siguiente iteración. A diferencia de la sentencia break, continue no termina completamente la ejecución del bucle.

```
do {

  i = i + 1;

  continue;

} while (i <5);
```

Sentencia break (romper)

La sentencia break se utiliza para salir de un bucle, switch. Break finaliza inmediatamente el código encerrado en un for, while, do-while o switch y transfiere el control a la siguiente sentencia.

```
do {

  i = i + 1;

  if (i == 3) break;

} while (i <5);
```

Búsquedas en JavaScript

Existen diferentes métodos que se pueden usar en JavaScript para buscar elementos dentro de un arreglo. El método a elegir depende del caso de uso particular, por ejemplo:

1. Obtener todos los elementos del arreglo que cumplen una condición específica. (**Filter**)

2. Obtener al menos uno de los elementos del arreglo que cumple dicha condición. (**Find**)
3. Obtener si un valor específico es parte del arreglo (**Includes**)
4. Obtener el índice de un valor específico (**IndexOf**).

Array.filter()

Podemos usar el método Array.filter() para encontrar los elementos dentro de un arreglo que cumplan con cierta condición. Por ejemplo, si queremos obtener todos los elementos de un arreglo de números que sean mayores a 10, podemos hacer lo siguiente:

```
let arreglo = [10, 11, 3, 20, 5];

let mayorQueDiez = arreglo.filter(element => element > 10);

console.log(mayorQueDiez) // resultado esperado: [11, 20]
```

Array.find()

Usamos el método Array.find() para encontrar el primer elemento que cumple cierta condición. Tal como el método anterior, toma un Callback como argumento y devuelve el primer elemento que cumpla la condición establecida. Usemos el método find en el arreglo del ejemplo anterior.

```
let arreglo = [10, 11, 3, 20, 5];

let existeElementoMayorQueDiez = arreglo.find(element => element > 10);

console.log(existeElementoMayorQueDiez) // resultado esperado: 11
```

Array.includes()

El método includes() determina si un arreglo incluye un valor específico y devuelve verdadero o falso según corresponda. En el ejemplo anterior, si queremos revisar si 20 es uno de los elementos del arreglo, podemos hacer lo siguiente:

```
let arreglo = [10, 11, 3, 20, 5];

let incluyeVeinte = arreglo.includes(20);

console.log(incluyeVeinte) // resultado esperado: true
```

Array.indexOf()

El método indexOf() devuelve el primer índice encontrado de un elemento específico. Devuelve -1 si el elemento no existe en el arreglo. Volvamos a nuestro ejemplo y encontremos el índice de 3 en el arreglo.

```
let arreglo = [10, 11, 3, 20, 5];

let indiceDeTres = arreglo.indexOf(3);

console.log(indiceDeTres) // resultado esperado: 2
```

Búsqueda de Máximos y Mínimos

Uno de los problemas académicos más comunes es el de la búsqueda del valor máximo o mínimo dentro de una lista. JavaScript dispone de las funciones Math.max() y Math.min() con las que es posible obtener el máximo y mínimo respectivamente de un conjunto de números, por ejemplo:

```
Math.max(1, 2, 3, 4, 5); // resultado esperado: 5

Math.min(1, 2, 3, 4, 5); // resultado esperado: 1
```

El problema de estas funciones es que no permiten entradas de tipo array, solamente de tipo numérico. Normalmente se puede solucionar empleando diferentes aproximaciones como son los métodos reduce() o apply(). La forma más fácil de aplicar una función a un array es utilizando el método apply(). Simplemente se tiene que aplicar apply() a la función pasando como primer parámetro null y como segundo parámetro el array. Así se puede obtener el máximo o mínimo de un array simplemente con el siguiente código.

```
Math.max.apply(null, values) // resultado esperado: 5

Math.min.apply(null, values) // resultado esperado: 1
```

Búsqueda Secuencial

La búsqueda secuencial se define como la búsqueda en la que se compara elemento por elemento del vector/array con el valor que buscamos. Es decir, un clásico recorrido secuencial (for). De hecho, tenemos varias maneras de implementarlo, pero más allá de eso, el principio es el mismo. Comparar elemento por elemento hasta encontrar el/los que buscamos. Este tipo de búsqueda en el

peor de sus casos ejecuta las instrucciones del loop n veces, es decir es la cantidad de elementos del arreglo X . En el mejor de los casos, el primer elemento del arreglo es el elemento que estamos buscando, por eso para ese caso ese elemento pasaría a ser X(1).

Veamos un ejemplo de una función que implementa búsqueda secuencial en un arreglo:

```
// Devolverá el índice donde encontró al elemento. Recibe el valor a buscar y el arreglo donde buscará

function sequentialSearch(element, array){

  for (var i in array){

    if (array[i] == element) return i;

  }

  return -1;

}

 var letters = ["a", "b", "c", "d", "f", "g", "h", "i", "j", "k", "l", "m", "n"];

sequentialSearch("g",letters);
```

Ordenamiento

Javascript provee un método que ordena los elementos de un arreglo localmente y devuelve el arreglo ordenado Array.prototype.sort([compareFunction(a, b)]). El modo de ordenación por defecto responde a la posición del valor del string de acuerdo a su valor en el juego de caracteres Unicode. Cuando se utiliza el método sort(), los elementos se ordenarán en orden ascendente (de la A a la Z) por defecto:

```
const equipos = ['Real Madrid', 'Manchester Utd', 'Bayern Munich', 'Juventus'];

equipos.sort();

// ['Bayern Munich', 'Juventus', 'Manchester Utd', 'Real Madrid']
```

Dada esta característica el ordenar números pasa a ser una tarea no tan simple. Si aplicamos el método sort() directamente a un arreglo de números, veremos un resultado inesperado:

```
const numeros = [3, 23, 12];

numeros.sort(); // --> 12, 23, 3
```

El método sort() puede ordenar valores negativos, cero y positivos en el orden correcto. Cuando compara dos valores, se puede enviar la función compareFunction(a, b), como función de comparación y luego se ordenan los valores de acuerdo al resultado devuelto:

1. Si el resultado es negativo, a se ordena antes que b.
2. Si el resultado es positivo, b se ordena antes de a.
3. Si el resultado es 0, nada cambia.

Por ende si queremos ordenar los números en orden ascendente, esta vez necesitamos restar el primero parámetro (a) del segundo (b):

```
var numbers = [4, 2, 5, 1, 3];

numbers.sort(function(a, b) {

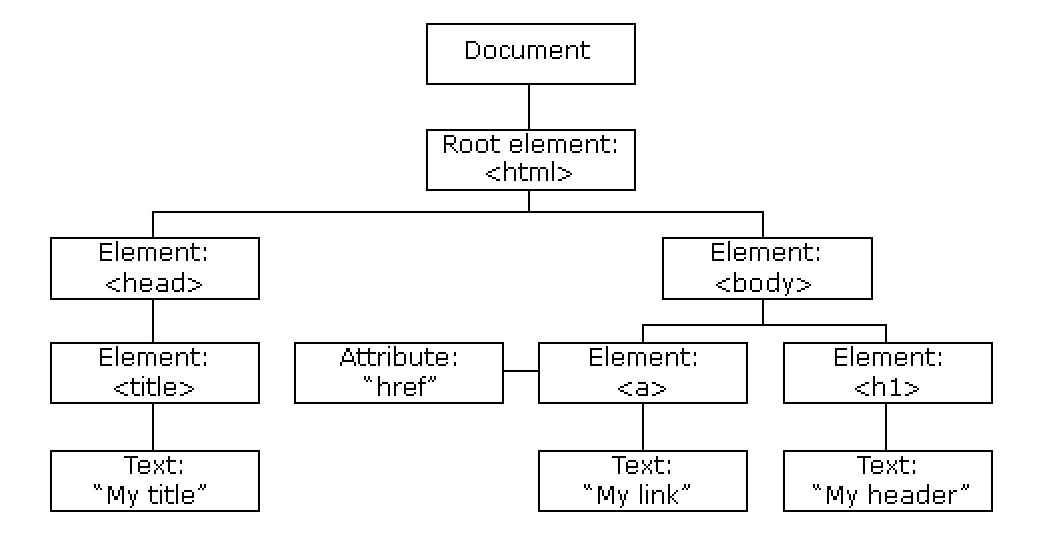
  return a - b;

}); // --> 1, 2, 3, 4, 5
```

¿DOM que es?

"*El Modelo de Objetos de Documento (DOM) del W3C es una plataforma e interfaz de lenguaje neutro que permite a los programas y scripts acceder y actualizar dinámicamente el contenido, la estructura y el estilo de un documento*"

Con HTML DOM, JavaScript puede acceder y cambiar todos los elementos de un documento HTML.



Tipos de nodos:

- **Document,** nodo raíz del que derivan todos los demás nodos del árbol.
- **Element,** representa cada una de las etiquetas HTML. Se trata del único nodo que puede contener atributos y el único del que pueden derivar otros nodos.
- **Attr,** se define un nodo de este tipo para representar cada uno de los atributos de las etiquetas HTML, es decir, uno por cada par atributo=valor.
- **Text,** nodo que contiene el texto encerrado por una etiqueta HTML.

Nota: Document representa la página web, por ende, para acceder a cualquier elemento de una web, se debe primero acceder a document.

Métodos

Los métodos son acciones que se pueden realizar a los elementos HTML mediante DOM.

Los más comunes son los utilizados para encontrar elementos:

1. Encontrar elementos HTML por ID (**getElementById**)
2. Encontrar elementos HTML por nombre de etiqueta (**getElementsByTagName**)
3. Encontrar elementos HTML por nombre de clase (**getElementsByClassName**)
4. Encontrar elementos HTML mediante selectores CSS (**querySelectorAll**)

Este ejemplo encuentra el elemento con id="intro":

```
const element = document.getElementById("intro");
```







Este ejemplo encuentra todos los <p>elementos:

```
const element = document.getElementsByTagName("p");
```

Este ejemplo devuelve una lista de todos los elementos con class="intro".

```
const x = document.getElementsByClassName("intro");
```

Este ejemplo devuelve una lista de todos los elementos <p> con class="intro".

```
const x = document.querySelectorAll("p.intro");
```

Elementos

Los elementos del DOM pueden ser creados, modificados o eliminados. A continuación veremos las principales acciones que se pueden realizar.

Cambiar elementos HTML:

Propiedad	Descripción
element.innerHTML = new html content	Cambia el contenido de un elemento HTML
element.attribute = new value	Cambia el valor del atributo de un elemento HTML
element.style.property = new style	Cambia el valor del atributo de un elemento HTML
Método	Descripción
element.setAttribute(attribute, value)	Cambia el estilo de un elemento HTML.

Agregar y eliminar elementos:

Método	Descripción
document.createElement(element)	Crea un elemento HTML
document.removeChild(element)	Elimina un elemento HTML
document.appendChild(element)	Agrega un elemento HTML
document.replaceChild(new, old)	Reemplaza un elemento HTML

Ejemplo:

```
function addElement () {
  // obtener el elemento div con id = "div_example"
  const existDiv = document.getElementById("div_example");
  // crear un nuevo elemento div
  const newDiv = document.createElement("div");
  // agregar el nuevo elemento div existente
  existDiv.appendChild(newDiv);
}
```

Archivos JSON

JSON (notación de objetos javascript) es un formato de intercambio de datos. Es muy parecido a un subconjunto de sintaxis JavaScript, aunque no es un subconjunto en sentido estricto. Aunque muchos lenguajes de programación lo soportan, JSON es especialmente útil al escribir cualquier tipo de aplicación basada en JavaScript, incluyendo sitios web y extensiones del navegador. Por ejemplo, es posible almacenar la información del usuario en formato JSON en una cookie o almacenar las preferencias de extensión en JSON en una cadena de valores de preferencias del navegador.

JSON es capaz de representar números, valores lógicos, cadenas, valores nulos, arreglos y matrices (secuencias ordenadas de valores) y objetos (mapas de cadena de valores) compuestos de estos valores (o de otras matrices y objetos). JSON no representa de manera nativa tipos de datos más complejos como funciones, expresiones regulares, fechas, y así sucesivamente (en objetos de fecha serializados por defecto como una cadena que contiene la fecha en formato ISO, al no hacerlo de ida y vuelta, la información no se pierde por completo). Si se necesita que JSON represente tipos de datos adicionales, se puede transformar los valores, ya que son serializados, o antes de su deserialización.

Los programadores conocemos estas convenciones utilizadas por JSON:

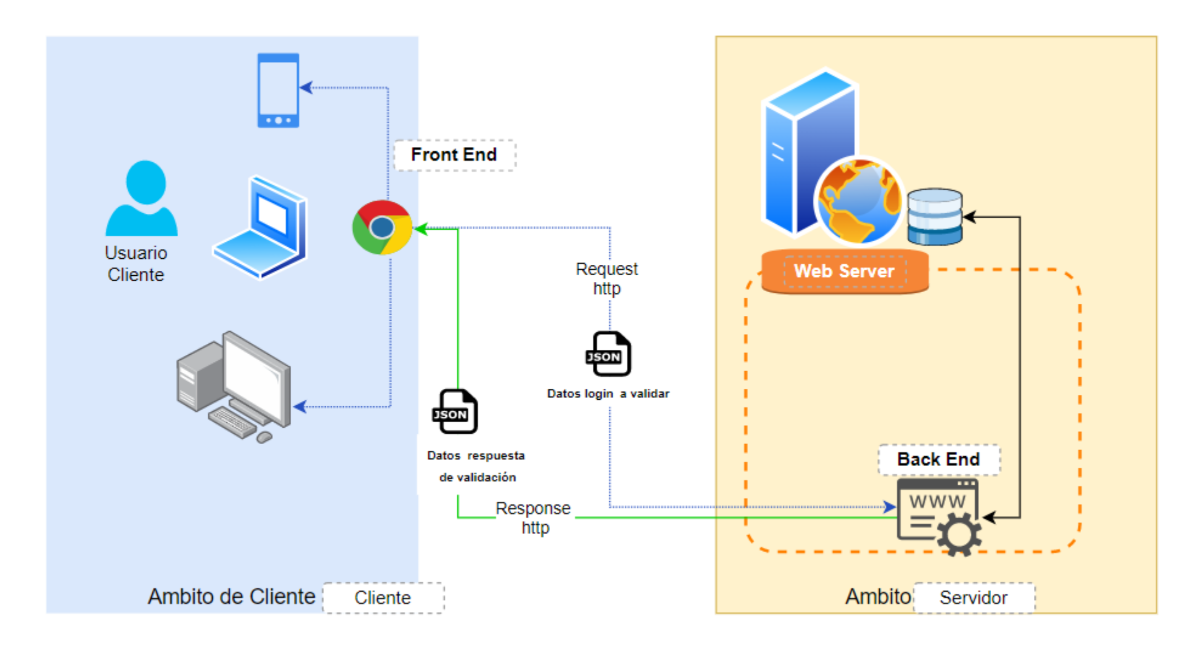
- - JSON son las siglas de JavaScript Object Notation.
- - El formato fue especificado por Douglas Crockford.
- - Fue diseñado para el intercambio de datos legibles por humanos.
- - Se ha ampliado desde el lenguaje de secuencias de comandos JavaScript.
- - La extensión del nombre de archivo es .json.
- - El tipo de Internet Media type es application / json.

¿Para qué se usa?

- El formato JSON se utiliza para serializar y transmitir datos estructurados a través de una conexión de red.
- Se utiliza principalmente para transmitir datos entre un servidor y aplicaciones web.
- Los servicios web y las API utilizan el formato JSON para proporcionar datos vía internet.
- Se puede utilizar con los lenguajes de programación y frameworks más actuales como Typescript, Java, Angular, Spring boot, entre muchos otros.

Nota: La mayoría de las bases de datos pueden almacenar los archivos JSON, aunque almacenar un JSON completo rompe la normalización de la base de datos.

En la siguiente imagen veremos un esquema que ilustra cómo viaja el archivo JSON:



Analizando la secuencia del esquema anterior, podemos observar la siguiente:

1. Un usuario ingresa sus datos de login en la página (Front End).
2. El código del Front End toma los datos y los serializa (convertir los datos/objeto del código a texto) en formato JSON y los envía al backend.
3. El Back End recibe el archivo JSON y lo deserializa (convertir el texto a objeto/código) para ser procesados, consultar la base de datos, etc.
4. Luego del procesamiento el Back End la respuesta es serializada (convertir los datos/objeto del código a texto) para enviársela al Front End notificando el resultado.

5. El código de la página Front End recibe el archivo JSON lo deserializa (convertir el texto a objeto/código) y muestra el resultado al usuario en la pantalla del navegador.

En la siguiente imagen te dejamos un ejemplo de cómo se ve un formato JSON en el código o en grilla:

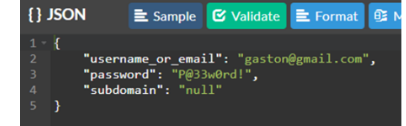
JSON representado en otras vistas

En código JavaScript

```
let text = '{ "username_or_email":"gaston@gmail.com" ,' +
' "password":"P@33w0rd!" ,' +
' "subdomain":"nul" }';
```

Cuando trabajamos en el código tenemos los datos en variables, objetos en la memoria. Para enviarlos debemos crear el formato JSON correcto, para ello podemos hacerlo manualmente o bien utilizar librerías que nos conviertan directamente los objetos a JSON.

En formato JSON



Este es el formato final JSON que viaja por la red hacia como si fuera un archivo de texto hacia otra aplicación.

En formato grilla



Esta es la representación del formato JSON en un formato grilla o un formato tabla. Veras que es muy parecido a como las bases de datos relacionales muestran sus datos en tablas.

Tipos de datos

El formato JSON admite los siguientes tipos de datos:







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

Nro	Tipo y descripción
1	**Number** formato de punto flotante de doble precisión en JavaScript
2	**String** Unicode entre comillas dobles con escape de barra invertida
3	**Boolean** verdadero o falso
4	**Array** una secuencia ordenada de valores
5	**Value** puede ser una cadena, un número, verdadero o falso, nulo, etc.
6	**Object** una colección desordenada de pares clave: valor
7	**Espacio en blanco** se puede utilizar entre cualquier par de tokens
8	**null** vacío

Reglas de sintaxis JSON:

- Los datos están en pares de nombre / valor
- Los datos están separados por comas
- Las llaves sostienen objetos
- Los corchetes sostienen matrices

Los datos en JSON: un nombre y un valor

Los datos JSON se escriben como pares de nombre / valor, al igual que las propiedades de los objetos de JavaScript. Un par de nombre / valor consta de un nombre de campo (entre comillas dobles), seguido de dos puntos, seguido de un valor:

"firstName":"Juana"

Los Objetos en JSON:





Los objetos JSON se escriben entre llaves. Al igual que en JavaScript, los objetos pueden contener varios pares de nombre / valor:

{"firstName":"Juana", "lastName":"Fernandez"}

Las Matrices en JSON

Las matrices JSON se escriben entre corchetes. Al igual que en JavaScript, una matriz puede contener objetos, te dejamos un ejemplo de una matriz Empleado con 3 Objetos:

```
"employees":[
  {"firstName":"Juana", "lastName":"Fernandez"},
  {"firstName":"Ana", "lastName":"Rntani"},
  {"firstName":"Francisco", "lastName":"Petrol"}
]
```

Validar formato JSON:

Existen muchas páginas para validar formato, te dejamos una para empezar, cada vez que trabajes con un formato JSON siempre debes estar seguro si el formato esta correcto y es válido, para ir a la página has clic aquí

Crear objetos simples de pruebas:

Los objetos JSON se pueden crear con JavaScript. Veamos las diversas formas de crear objetos JSON usando JavaScript:

- Creación de un objeto vacío

```
var JSONObj = {};
```

- Creación de un nuevo objeto -

```
var JSONObj = new Object();
```

- Creación de un objeto con atributo **nombrelibro** con valor en cadena, atributo **precio** con valor numérico. Se accede al atributo usando '.' Operador -

```
var JSONObj = { "nombrelibro":"El hacedor", "precio":500 };
```

TypeScript

Introducción

En esta sección introduciremos el lenguaje de programación Typescript, la sintaxis que añade, el tipado, genéricos, decoradores, etc. Repasaremos, además, algunos conceptos vistos previamente en el módulo de JavaScripts a fin de comprender mejor la utilización del mismo y finalmente, profundizaremos el paradigma de programación orientada a objetos (POO) siempre orientandonos a la utilización del lenguaje para su aplicación en Angular.

Objetivos

- Comprender la diferencia entre Javascript y Typescript.
- Reconocer y escribir código typescript reconociendo los tipos de datos y las inferencias que realiza el compilador TSC.
- Comprender la importancia de la programación orientada a objetos y su diferencia con la programación estructurada. Comprender los fundamentos de la programación orientada a objetos.
- Crear código Typescript en base a los principios de la programación orientada a objetos para resolver problemas.

TypeScript

Typescript es un lenguaje de programación de código abierto creado por el equipo de Microsoft como una solución al desarrollo de aplicaciones de gran escala con Javascript dado que este último carece de clases abstractas, interfaces, genéricos, etc. y demás herramientas que permiten los lenguajes de programación tipados. Son ejemplos la compatibilidad con el intellisense, la comprobación de tiempo de compilación, entre otras.

A typescript se lo conoce además como un superset (superconjunto) de JavaScript ya que es un lenguaje que transpila el fuente de un lenguaje a otro pero, incluye la ventaja de que es verdaderamente orientado a objetos y ofrece además, muchas de las cosas con las que estamos habituados a trabajar los desarrolladores como por ejemplo: interfaces, genéricos, clases abstractas, modificadores, sobrecarga de funciones, decoradores, entre otras varias.

"*Los superset compilan en el lenguaje estándar, por lo que el desarrollador programa en aquel lenguaje expandido, pero luego su código es "transpilado" para transformarlo en el lenguaje estándar, capaz de ser entendido en todas las plataformas*" (desarrolloweb.com)

Entonces, si posees algo de conocimiento de Javascript, ¡tienes ventaja! Podemos cambiar los archivos de JavaScript a TypeScript de a poco e ir preparando la base del conocimiento para incorporar en su totalidad este nuevo lenguaje.

JavaScript	TypeScript
JavaScript se ejecuta en el navegador. Es un lenguaje de programación interpretado.	TypeScript necesita ser “transpilado” a JavaScript, que es el lenguaje entendido por los navegadores.
JavaScript se ejecuta en el navegador, es decir en el lado del cliente únicamente.	TypeScript se ejecuta en ambos extremos. En el servidor y en el navegador.
Es débilmente tipado.	Es fuertemente tipado (tipado estático)
Basado en prototipos.	Orientado a objetos.

Tabla comparativa de los lenguajes JavaScripts y TypeScripts

Tipado estático

Una de las principales características de typescript es que es fuertemente tipado por lo que, no sólo permite identificar el tipo de datos de una variable mediante una sugerencia de tipo sino que además permite validar el código mediante la comprobación de tipos estáticos. Por lo tanto, TypeScript permite detectar problemas de código previo a la ejecución cosa que con Javascript no es posible.

Sugerencias para la escritura: Los tipos también potencian las ventajas de inteligencia y productividad de las herramientas de desarrollo, como IntelliSense, la navegación basada en símbolos, la opción Ir a definición, la búsqueda de todas las referencias, la finalización de instrucciones y la refactorización del código, podes consultar la documentación oficial acá.

Se agrega además que Typescript permite describir mucho mejor el fuente ya que, además de ser tipado, es verdaderamente orientado a objetos (avanzaremos en esto más adelante) lo que permite a los desarrolladores un código más legible y mantenible.

Variables

Antes de avanzar en tipos y subtipos de datos en Typescript, recordemos el concepto de variable:

Una **variable** es un espacio de memoria que se utiliza para almacenar un valor durante un tiempo (scope) en la ejecución del programa. La misma tiene asociado un tipo de datos y un identificador.

Debido a que Typescript es un superset de Javascript, la declaración de las variables se realiza de la misma manera que en Javascript:

var: Es el tipo de declaración más común utilizada.

En este punto es importante agregar que, si bien TypeScripts es muy flexible y puede determinar el tipo de datos implícitamente, es aconsejable utilizar la nomenclatura que recomienda la página oficial (tipado estricto), ya que de este modo permitirá un mejor mantenimiento de nuestro código y el mismo será más legible.

Sintaxis:

```
var medida= 10;
var m=10;
```

let: Es un tipo de variable más nuevo (agregado por la ECMAScript 2015). El mismo reduce algunos problemas que presentaba la sentencia var en las versiones anteriores de Javascript.

Este cambio permite declarar variables con ámbito de nivel de bloque y evita que se declare la misma variable varias veces, puedes profundizar en la documentación oficial haciendo clic acá

Sintaxis:

```
let precio=0;
```

const: *Es un tipo constante ya que, al asumir un valor no puede modificarse (agregado también por la ECMAScript 2015)*

Sintaxis:

```
const ivaProducto = 0.10;
```

Nota: Como recordatorio, la diferencia entre ellas es que las declaraciones ***let*** se pueden realizar sin inicialización, mientras que las declaraciones ***const*** siempre se

inicializan con un valor. Y las declaraciones ***const***, una vez asignadas, nunca se pueden volver a asignar. (Referencia).

Inferencia de tipo en TypeScript

Typescript permite asociar tipos con variables de manera explícita o implícita como veremos a continuación:

Sintaxis de la inferencia explícita:

```
<variable>: <tipo de datos>
```

Ejemplo inferencia explícita:

```
let edad: number; = 42;
```

Ejemplo inferencia implícita:

```
let edad = 42;
```

Nota: Si bien las asociaciones de tipo explícitas son opcionales en TypeScript, se recomiendan dado que permiten una mejor lectura y mantenimiento del código.

Veamos cómo funciona:

1. Abrir VSCode y crear un nuevo archivo titulado ***example01.ts***
2. En dicho archivo, escribir las siguientes declaraciones de variables:

```
let a: number;   //* Inferencia explícita
let b: string;   //* Inferencia explícita
let c=101;       //* Inferencia implícita
```

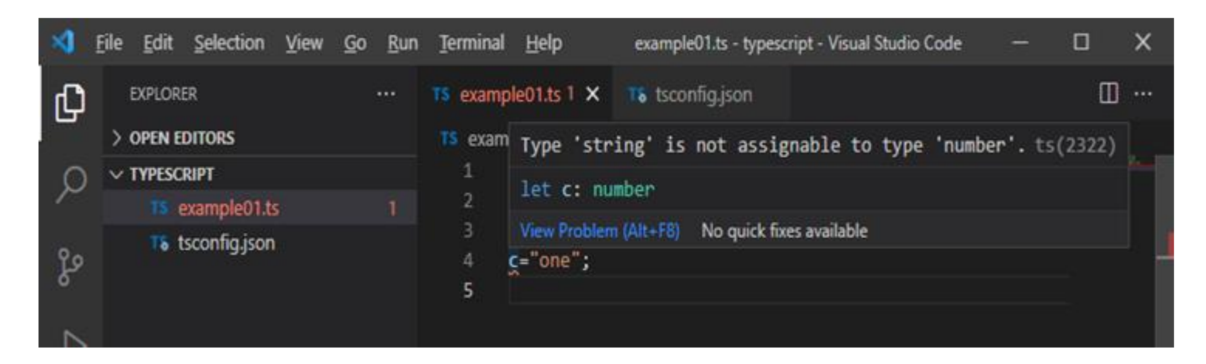
En este caso, typescript interpreta que la variable **a** es del tipo number y **b** del tipo string dado que la declaración es explícita. En el caso de la variable **c** infiere que es del tipo number dado que éste es el tipo de datos que corresponde al valor con el que se ha inicializado la variable.

Nota: Observa que al posicionar el puntero del mouse sobre la variable c, VSCode abre un tooltip con la declaración explícita "***let c:number***"

Pero ¿qué ocurre si intentamos asignar un tipo de datos diferente a la variable c?. Para ello, escribir a continuación la siguiente línea:

```
c="one";
```

VSCode marca un error en la línea de la asignación y en el explorador puedes ver el archivo en rojo. Observa además que al posicionar el puntero del mouse sobre la línea VSCode muestra el mensaje de error ***"Type string is not assignable to type number"***



Error typescript. Asignación de tipos.

Analicemos otro ejemplo, dada la siguiente declaración:

```
let recursos: ['memoria', 'disco','procesador'];
```

Hasta aquí sabemos que: "let", es la declaración de un arreglo cuyo nombre es "recursos" y el tipo no ha sido definido explícitamente.

```
'recursos' is declared but its value is never read. ts(6133)
let recursos: string[]
Quick Fix... (Ctrl+.)
let recursos= ['memoria','disco','procesador']
```

Entonces, si posicionamos el puntero del mouse sobre la variable "recursos" podemos observar que Typescripts automáticamente entiende por sí solo que se trata de un arreglo del tipo String.

Sin embargo, si luego introducimos en el array un valor de otro tipo, y posicionamos el puntero del mouse sobre la variable recursos, podremos observar que el arreglo admite variables del tipo "string" o (símbolo para "o" es "|") "number".

```
'recursos' is declared but its value is never
read. ts(6133)
let recursos: (string | number)[]
Quick Fix... (Ctrl+.)
let recursos= ['memoria','disco',100]        You, se
```

Y quizás, este no sea el comportamiento que deseamos por ello, se aconseja como buena práctica especificar el tipo de datos de manera explícita.

En este caso:

```
let recursos: string [] = ['memoria','disco','procesador']
```

De esta manera, si intentamos introducir un elemento number u otro tipo a nuestro arreglo, el compilador nos marcará un error.

Iniciando con TypeScript

¿Cómo instalar TypeScrypt?

Para ejecutar código en javascript necesitamos instalar:

- NodeJS, dado que el compilador de Typescript está desarrollado en NodeJS.

- TSC (Command-line TypeScript Compiler), herramienta que permite compilar un archivo TypeScript a Javascript nativo.

¿Cómo instalar NodeJS?

Para ello, seguir los siguientes pasos:

1. Descargar del sitio oficial el instalador acorde a su sistema operativo.
2. Instalar NodeJs.
    a. Si tienes Windows o Mac, simplemente ejecuta el instalador y ¡listo!
    b. Si tienes Linux, la página de instalación de NodeJS ofrece los comandos para instalar en Linux. Es relativamente sencillo.
3. Evaluar que la instalación fue exitosa. Para ello, ir a la línea de comandos del sistema e introducir el comando: ***node***. A continuación, el sistema mostrará por pantalla la versión de NodeJS instalada como sigue:

```
Símbolo del sistema - node
Microsoft Windows [Versión 10.0.19041.1083]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\ivana>node
Welcome to Node.js v14.16.0.
Type ".help" for more information.
>
```

Línea de comandos del sistema Windows después de ejecutar el comando "node".

¿Cómo instalar TSC (Command-line TypeScript Compiler)?

La misma se realizará vía comando mpn como sigue:

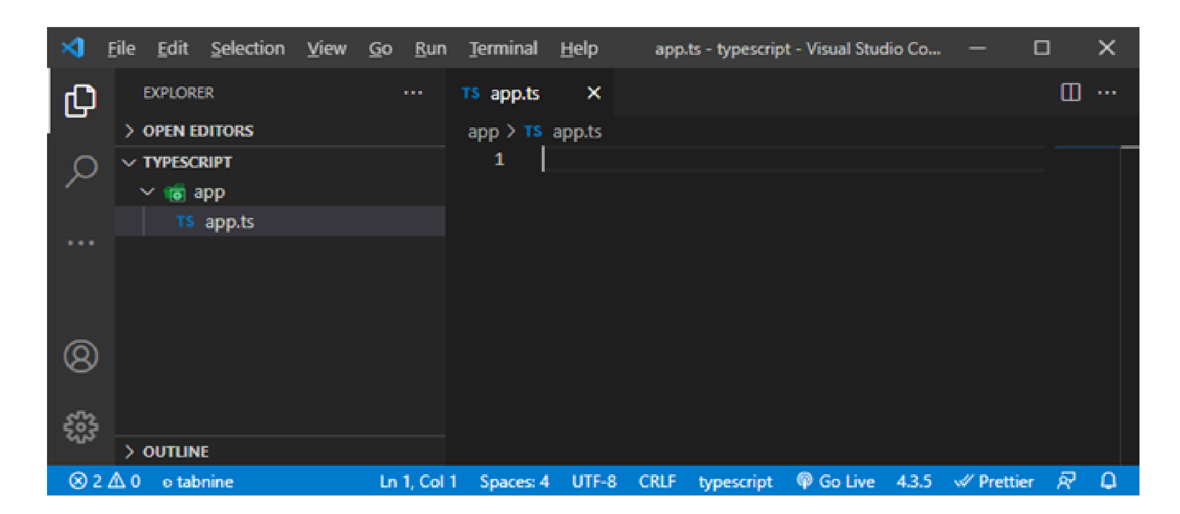
1. Abrir la línea de comandos del sistema.
2. Ejecutar el siguiente comando: ***npm install -g typescript***
3. Evaluar que la instalación fue exitosa. Para ello ejecutar el comando: ***tsc -v***



Línea de comandos del sistema luego de ejecutar el comando tsc -v

¿Cómo crear y compilar un archivo Typescript en VSCode?

1. Desde el explorador de archivos, crear un nuevo archivo titulado: ***app.ts*** dentro de una carpeta app (opcional) como sigue:

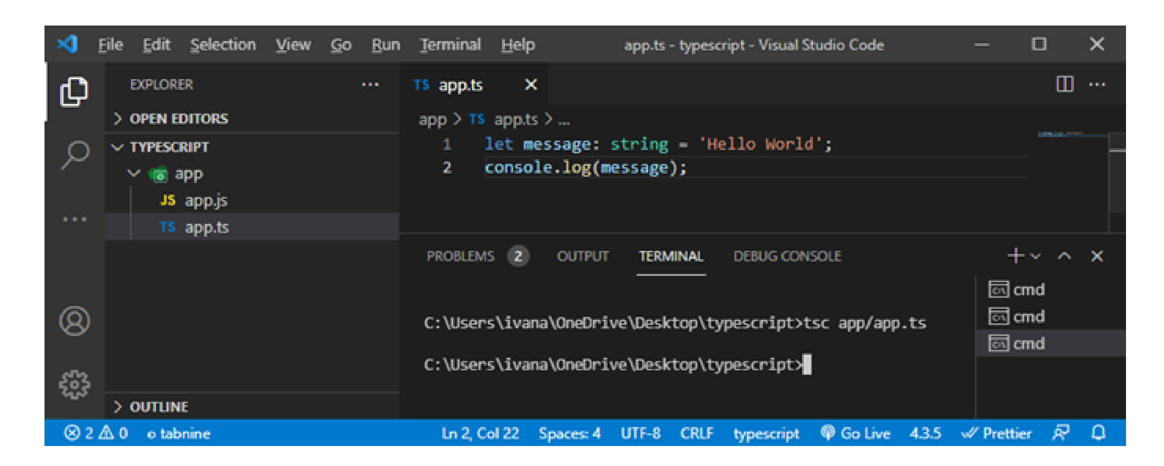


Creación del archivo helloworld.ts

2. Luego, escribir el código typescript en dicho archivo:

```
let message: string = 'Hello World';
console.log(message);
```

3. Finalmente, haciendo uso de la terminal de VSCode ejecutar el comando: ***tsc app/app.ts.*** Este comando compila y si no hay ningún error, crea el nuevo archivo js.



Compilación de Typescript a Javascript

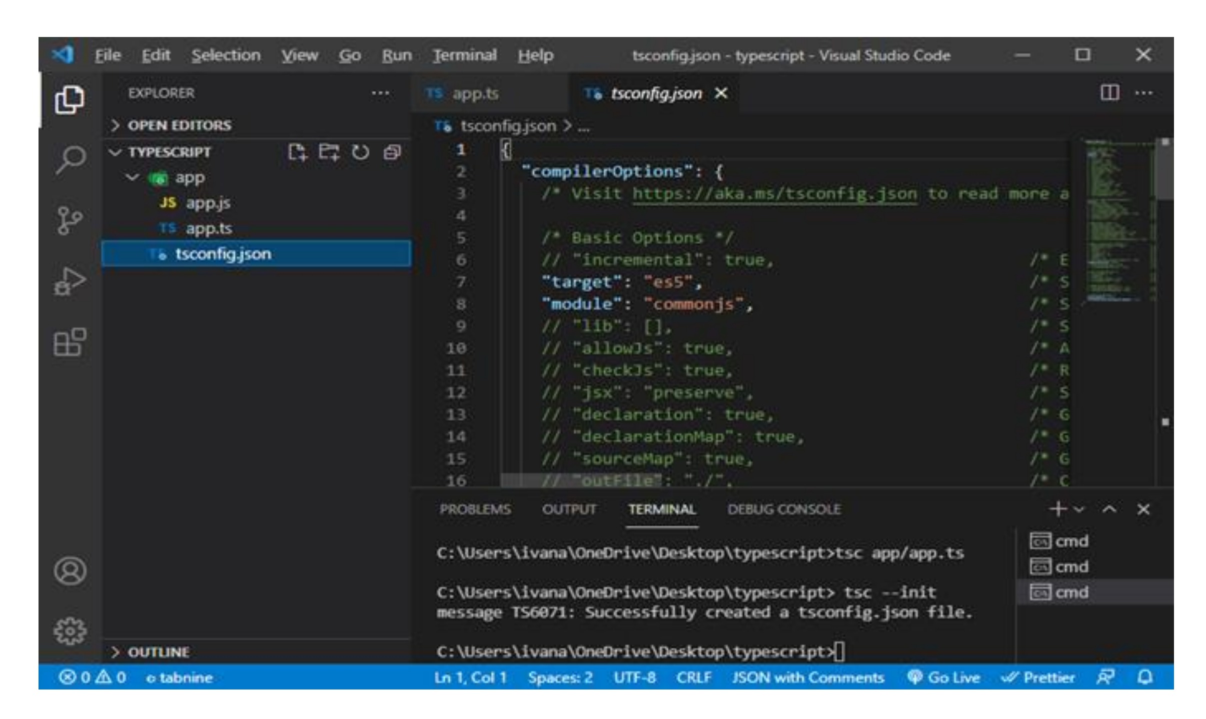
Comandos y opciones del compilador CLI de tsc

"Las opciones del compilador permiten controlar cómo se genera el código JavaScript a partir del código TypeScript de origen. Puedes establecer las opciones en el símbolo del sistema, como haría en el caso de muchas interfaces de la línea de comandos, o en un archivo JSON denominado tsconfig.json.

Hay disponibles numerosos comandos para las opciones del compilador por línea de comandos. Puedes ver la lista de opciones el sitio oficial haciendo clic acá, también te dejamos más información sobre la compilación de TypeScript que puedes leer haciendo clic acá

Para modificar el comportamiento predefinido del TSC con VSCode:

1. Abrir la terminal.
2. Ejecutar el comando: ***tsc --init.*** A continuación, se creará el archivo ***tsconfig.json***



Archivo tsconfig.json generado después de ejecutar el comando tsc --init

3. Editar las configuraciones según se requiera.

Por ejemplo, podemos crear una carpeta que contenga todos los archivos .js generados por el compilador TSC (el output dir/file). Para ello, descomentar la entrada "outFile" y a continuación ejecutar como sigue:

```
"outFile": "./output/app.js",
```

y comentamos la entrada "module":

```
//"module": "commonjs",
```

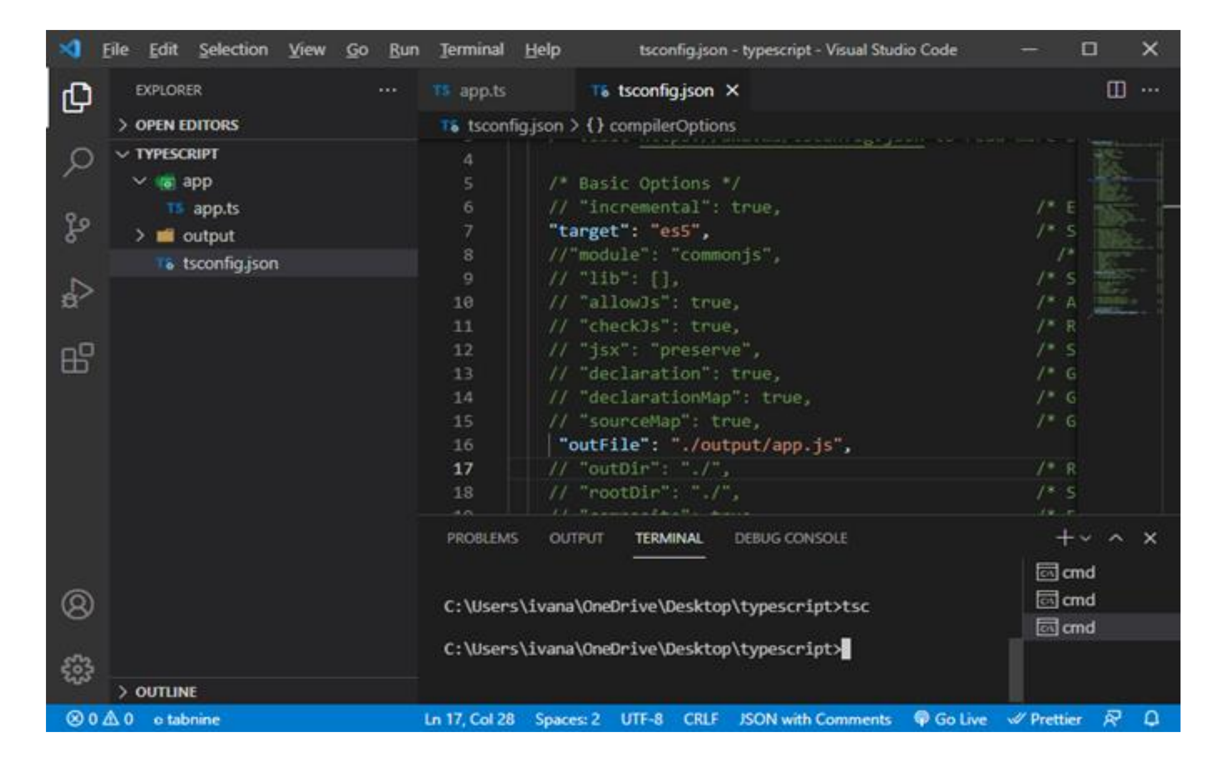






2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

y finalmente, ejecutar en la terminal de VSCode el comando ***"tsc"***. A continuación, se creará la carpeta **output** conteniendo el archivo **app.js**.

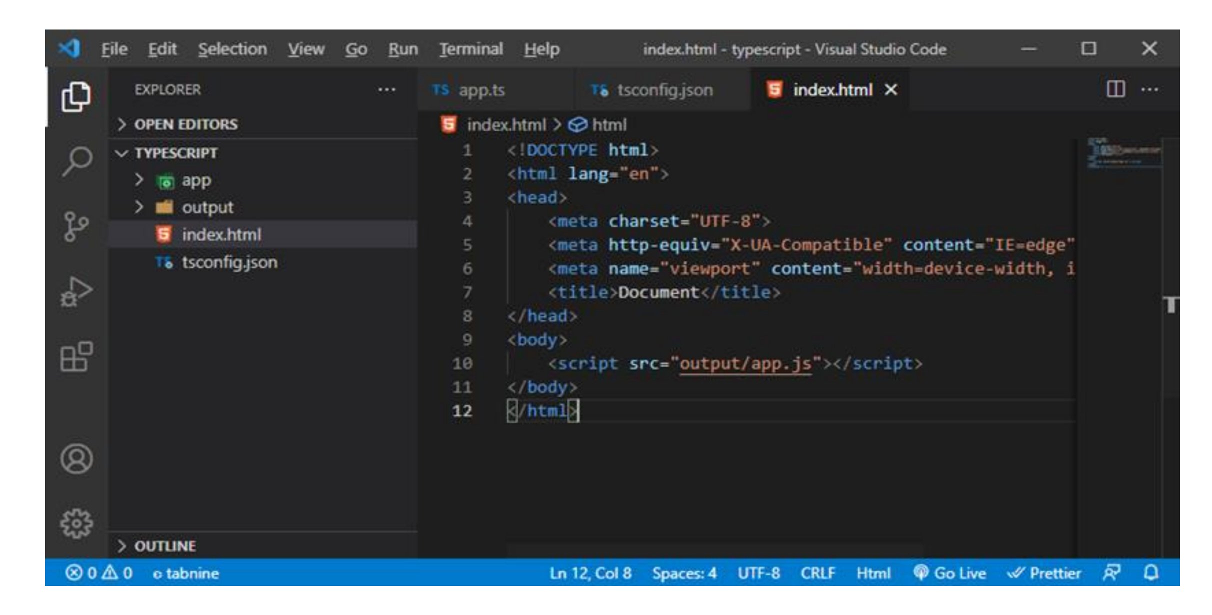


Manipulando la configuración del TSC para que tire los archivos generados a una carpeta output.

Hasta el momento hemos creado un archivo **app.ts** y lo hemos transpilado a un archivo equivalente en javascript **app.js** usando el compilador TSC pero, *¿cómo podemos ejecutarlo para ver los resultados en la consola?*

Para ello, realizar los siguientes pasos:

1. Crear un archivo html que incluya el script app.js:



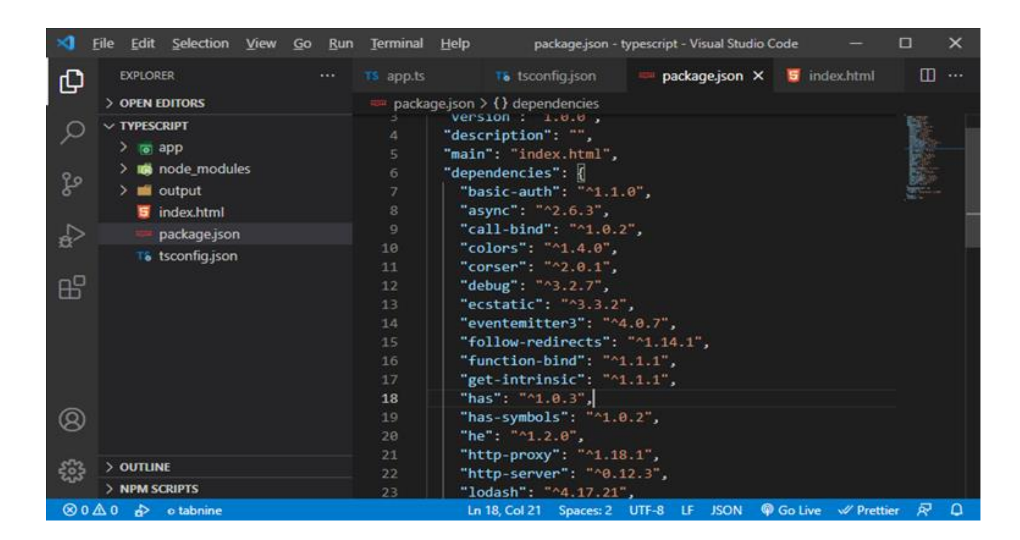
Archivo index.html

2. Ejecutar el archivo index.html o configurar un servidor de prueba para el entorno de desarrollo.

¿Cómo crear un servidor de pruebas en VSCode?

Para ello, seguir los siguientes pasos:

1. Ejecutar el siguiente comando ***"npm install --global http-server"***. A continuación, se creará la carpeta ***node_modules***.
2. Ejecutar el comando ***"npm init"***. A continuación, se creará un archivo ***package.json***



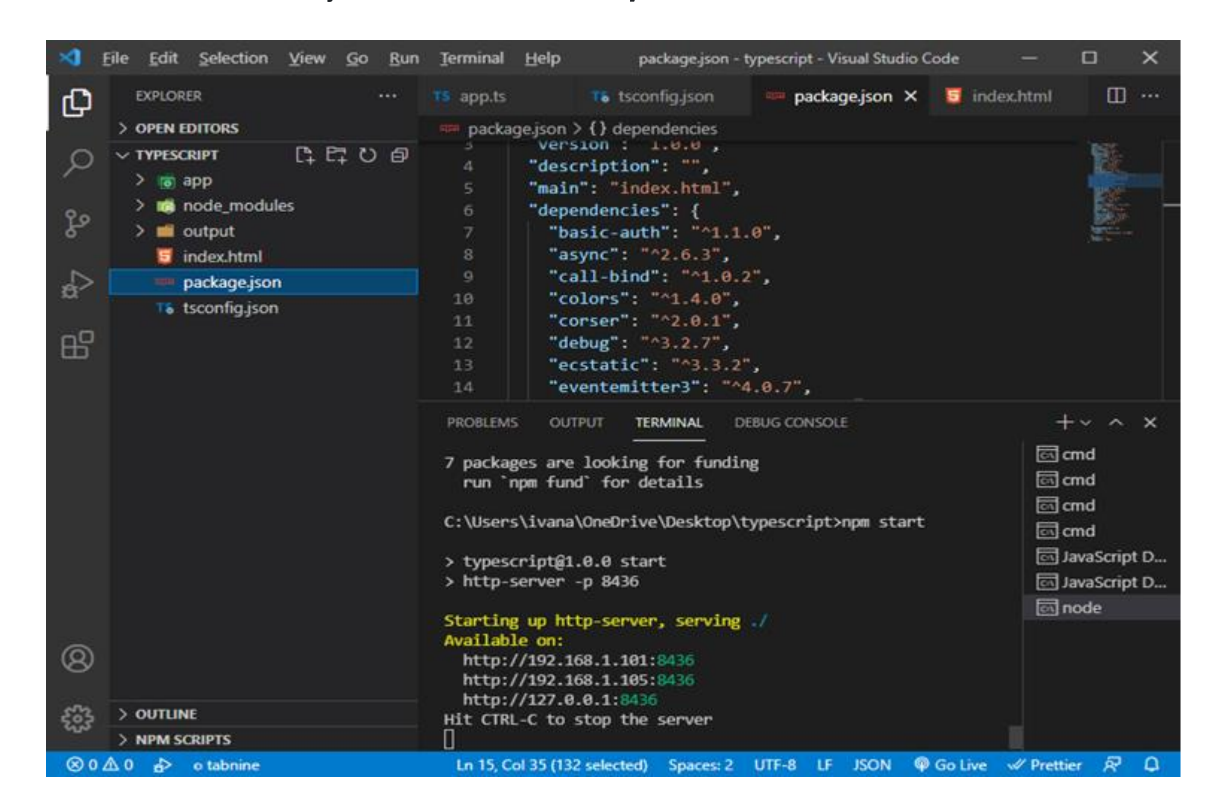
Archivo package.json

Nota: El archivo package.json es un archivo que contiene todos los metadatos acerca del proyecto. Son ejemplos: descripción, licencia, autor, dependencias, scripts, entre otros.

3. Configurar la entrada "scripts" como sigue:

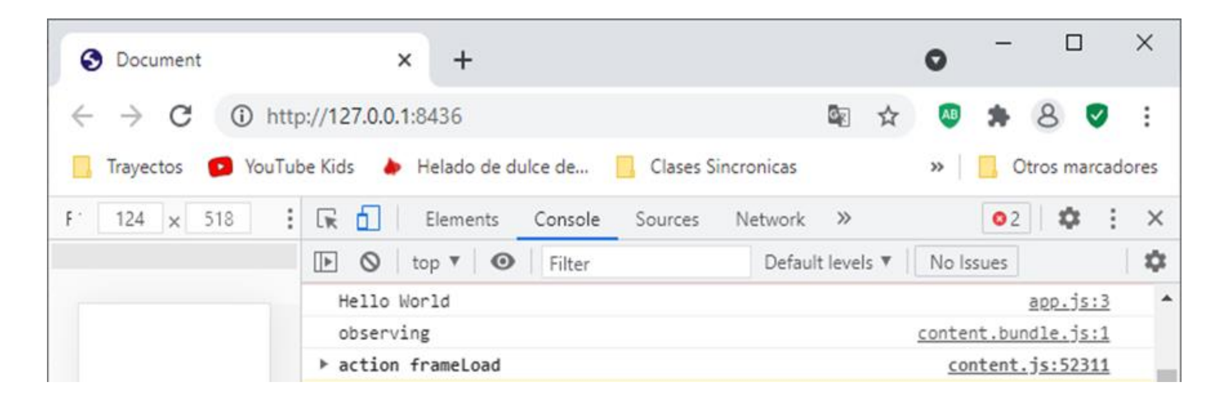
```
"scripts": {
  "start": "http-server -p 8456"
}
```

4. Finalmente, ejecutar el comando ***"npm-start"*** :



Iniciando el servidor.

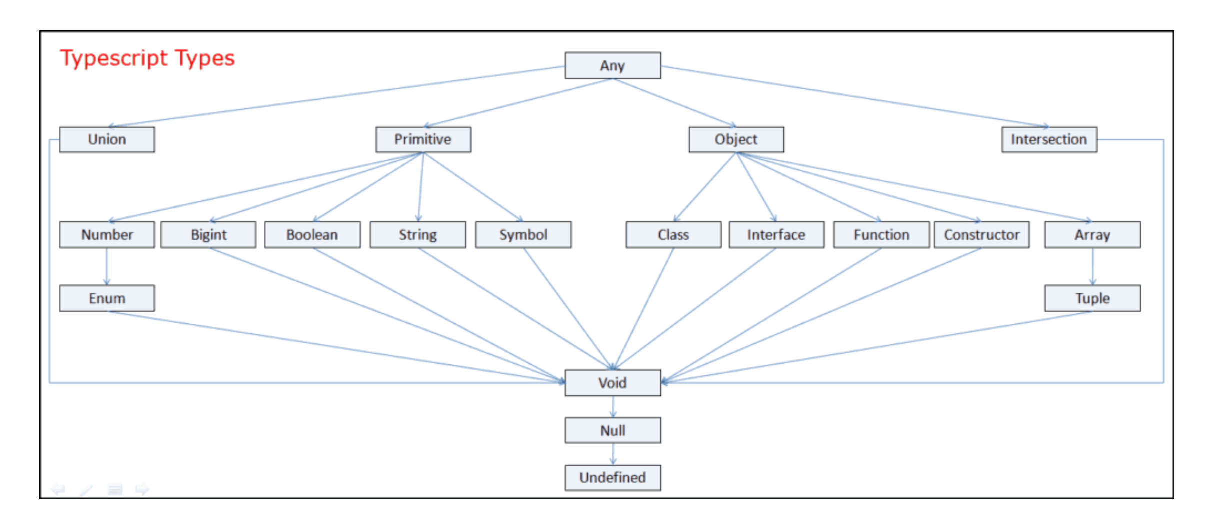
Como podemos observar en la Terminal de VSCode, accediendo a la url: http://127.0.0.1:8436 podremos visualizar nuestro html y, si inspeccionamos el fuente el mensaje "Hola Mundo" en la consola.



Inspección de código index.html

Tipos de datos y subtipos

Todos los tipos en TypeScript son subtipos de un único tipo principal denominado tipo any. Any es un tipo que representa cualquier valor de JavaScript sin restricciones.



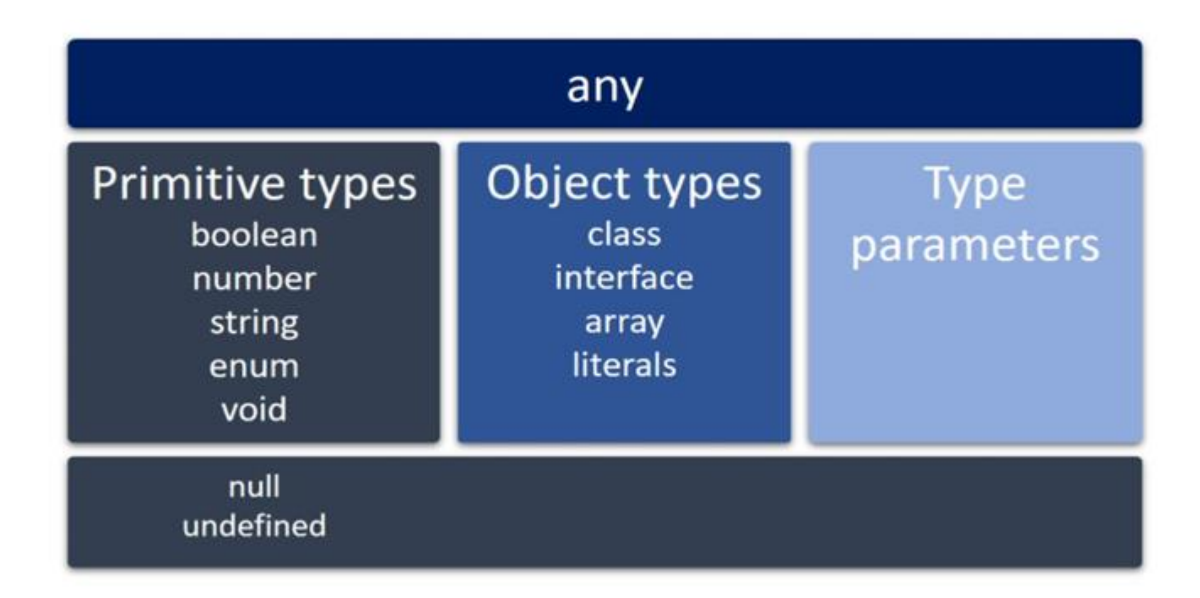
Any: Puede ser de cualquier tipo y su uso está justificado cuando no tenemos información a priori de qué tipo de dato se trata. Este tipo de definición es propia de TypeScripts.

Sintaxis:

```
let cantidad: any= 4;

let desc: any [] =[1,true,"verde"]
```

Todos los demás tipos se clasifican como primitivos, de objeto o parámetros. Estos tipos presentan diversas restricciones estáticas en sus valores.



Overview of types in TypeScript - Learn. (s. f.). Microsoft Docs. Recuperado 11 de octubre de 2021, de https://docs.microsoft.com/en-us/learn/modules/typescript-declare-variable-types/2-types-overview

Tipos de datos primitivos

Los tipos primitivos son: boolean, number, string, void, null, undefined y enum.

string: Representa valores de cadena de caracteres (letras);

Sintaxis:

```
let saludo: string = "hola, mundo";
```

TypeScript permite también usar plantillas de cadenas con las que podemos intercalar texto con otras variables: ${ expr }

Ejemplo:

```
let nombre: string = "Mateo";
let mensaje: string = `Mi nombre es ${nombre}.
    Soy nuevo en Typescript.`;
console.log(mensaje);
```

number: Representa valores numéricos, como enteros (int) o decimales (float).

Sintaxis:

```
let codigoProducto: number = 6;
```

boolean: *Es un tipo de variable que puede tener solo dos valores, Verdadero (true) o Falso (false).* Sintaxis:

```
let estadoProducto: boolean = false;
```

Void: *El tipo void existe únicamente para indicar la ausencia de un valor, como por ejemplo en una función que no devuelve ningún valor.*
Sintaxis:

```
function mensajeUsuario(): void {
 console.log("Este es un mensaje para el usuario");
}
```

Enum: *Las enumeraciones ofrecen una manera sencilla de trabajar con conjuntos de constantes relacionadas. Un elemento enum es un nombre simbólico para un conjunto de valores. Las enumeraciones se tratan como tipos de datos y se pueden usar a fin de crear conjuntos de constantes para su uso con variables y propiedades.*
Siempre que un procedimiento acepte un conjunto limitado de variables, considere la posibilidad de usar una enumeración. Las enumeraciones hacen que el código sea más claro y legible, especialmente cuando se usan nombres significativos” (referencia)

Ejemplo:

```
//**Crear la enumeración */
enum Color {
    Blanco,
    Rojo,
    Verde
}

//**Declarar la variable y asignar un valor de la enumeración */
let colorAuto: Color= Color.Blanco;

console.log(colorAuto); //return 0
```

Tipos de objetos

Los tipos de objeto son todos los tipos de clase, de interfaz, de arreglos y literales.

<u>Nota</u>: Los tipos de clase e interfaz se abordarán más adelante en este mismo módulo.

Array: *Es un tipo de colección o grupos de datos (vectores, matrices). El agrupamiento lleva como antecesor el tipo de datos que contendrá el arreglo.*

Sintaxis:

```
let list : string[]=['pimiento','papas','tomate'];
let rocosos:  boolean[] = [true, false, false, true]

let perdidos: any[] = [9, true, 'asteroides'];

let diametro: [string, number] = ['Saturno', 116460];
```

Generic.*: También puedes definir tipos genéricos como sigue.* Sintaxis:

```
function identity<T>(arg: T): T {
  return arg;
}
```

Los genéricos son como una especie de plantillas mediante los cuales podemos aplicar un tipo de datos determinado a varios puntos de nuestro código.

Sirven para aprovechar código, sin tener que duplicarlo por causa de cambios de tipo y evitando la necesidad de usar el tipo "any".

Los mismos se indican entre "mayores y menores" y pueden ser de cualquier tipo incluso clases e interfaces.

Veamos el ejemplo que nos provee la fuente oficial de typescript:

Si tenemos la siguiente función:

```
function identity(arg: number): number {

  return arg;

}
```

Pero, necesitamos que la misma sea válida para otros tipos de datos entonces podríamos cambiar el tipo number por any como sigue:

```
function identity(arg: any): any {

  return arg;

}
```

Sin embargo, el tipo any permite cualquier tipo de valor por lo que la función podría recibir un tipo number y devolver otro. Entonces, estamos perdiendo información sobre el tipo que debe devolver la función. Para solucionarlo, y obligar al compilador que respete el mismo tipo (parámetros de entrada y salida) podemos utilizar genéricos.

```
function identity<T>(arg: T): T {

  return arg;

}
```

Observa que cambiamos any por la letra **T**.

T nos permite capturar el tipo de datos por lo que el tipo utilizado para el argumento es el mismo que el tipo de retorno.

Object: *Es un tipo de dato que engloba a la mayoría de los tipos no primitivos.*

```
let persona:object={nombre:"Ana", edad:45}
```

Desestructuración: *La desestructuración permite acceder a los valores de un array o un objeto.*

Ejemplo - desestructuración de un objeto:

```
var obj={a:1,b:2,c:3};
console.log(obj.c);
```

Ejemplo - desestructuración de un array:

```
var array=[1,2,3];
console.log(array[2]);
```

Ejemplo - desestructuración con estructuración:

```
var array=[1,2,3,5];
var [x,y, ...rest]= array;
console.log(rest);
```

Observa que la sintaxis ***...rest***, nos permite agregar más parámetros. En este caso el resultado en consola será: [3, 5]

Estructuración: *Como se pudo observar en el apartado anterior, la estructuración facilita que una variable del tipo array reciba una gran cantidad de parámetros*.

Ejemplo en funciones:

```
function rest(first, second, ...allOthers)

{console.log(allOthers);
}
```

Observa que la sintaxis ***...allOthers*** nos permite pasar más parámetros.

Luego, al llamar a la función con los siguientes parámetros:

```
rest('1', '2','3','4','5'); //return 3,4,5
```

Tipos null y undefined

"*Los tipos null y undefined son subtipos de todos los demás tipos. No es posible hacer referencia explícita a los tipos null y undefined. Solo se puede hacer referencia a los valores de esos tipos mediante los literales null y undefined*"

Aserción de tipos (As)

Una aserción de tipos le indica al compilador "***confía en mí, sé lo que estoy haciendo***". Se parece al *casting* en otros lenguajes de programación, pero no tiene impacto en tiempo de ejecución sino que le dice al compilador el tipo de datos en cuestión a fin de acceder a los métodos, propiedades, etc. del tipo de datos en tiempo de desarrollo.

Sintaxis (dos posibles):

```
// primera posibilidad
(nombre as string).toUpperCase();

// primera posibilidad
(<string>nombre).toUpperCase();
```

Funciones

Una función es un **conjunto de instrucciones** o sentencias que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente y se caracterizan porque:

- deben ser invocadas por su nombre.
- permiten **simplificar el código** haciendo más legible y reutilizable.

La declaración de una función consiste en:
- Un nombre
- Una lista de parámetros o argumentos encerrados entre paréntesis.
- Conjunto de sentencias o instrucciones encerradas entre llaves.

Sintaxis:

```
function nombre (parámetro1, parámetro2)
{
//**instrucciones a ejecutar */

}
```

Ejemplo:

```
function calcularIva (produtos:Producto[]):[number, number]{
  let total=0;
  produtos.forEach(({precio}) =>{
    total+= precio;
  });
  return [total, total*0.15];

}

// Clase Producto
class Producto {
  precio:number;

}
```

Nota: Puedes crear tus propias funciones y usarlas cuando sea necesario.

TypeScrypt Programación Orientada a Objetos

A continuación abordaremos el tema de programación orientada a objetos ya que son de suma importancia para trabajar con Typescript y consecuentemente en Angular.

La programación orientada a objetos (abreviada de ahora en más como POO), es un conjunto de reglas y principios de programación (o sea, un paradigma de programación) que busca representar las entidades u objetos del dominio (o enunciado) del problema dentro de un programa, de la forma más natural posible.

En el paradigma de programación tradicional o *estructurado*, que es el que hemos usado hasta aquí, el programador busca identificar los *procesos* (en forma de subproblemas o módulos) a fin de obtener los resultados deseados. Y esta forma de proceder no es en absoluto incorrecta: la estrategia de descomponer un problema en subproblemas es una técnica elemental de resolución de problemas que los programadores orientados a objetos siguen usando dentro de este nuevo paradigma. Entonces, ¿a qué viene el paradigma de la POO?

La programación estructurada se basa en descomponer procesos en subprocesos y programando cada uno como ***rutina*, *función*, o *procedimiento*** (todas formas de referirse al mismo concepto) Sin embargo, esta forma de trabajar resulta difícil al momento de desarrollar sistemas *realmente* grandes o de mucha complejidad. Es decir que, la sola orientación a la descomposición en subproblemas no alcanza cuando el sistema es complejo dado que se vuelve difícil de visualizar su estructura general, se hace complicado realizar hasta las más pequeñas modificaciones sin que estas reboten en la lógica de un número elevado de otras rutinas, y por último es casi imposible replantear el sistema para agregar nuevas funcionalidades que permitan que ese sistema simplemente siga siendo útil frente a continuas nuevas demandas.

La *POO* significa una nueva visión en la forma de programar, buscando aportar claridad y naturalidad en la manera en que se plantea un problema. Ahora, el objetivo primario no es identificar procesos sino identificar *actores*:
las *entidades* u *objetos* que aparecen en el escenario o dominio del problema, tales que esos objetos tienen no sólo datos asociados sino también algún comportamiento que son capaces de ejecutar. Por ejemplo, pensemos en un *objeto* como en un *robot virtual*: el programa tendrá muchos robots virtuales (objetos de software) que serán capaces de realizar eficiente y prolijamente ciertas tareas en las que serán expertos. Además, serán capaces de interactuar con otros robots virtuales (objetos...) con el objeto de resolver el problema que el programador esté planteando.

Ventajas de la POO

- Es una forma más natural de modelar.
- Permite manejar mejor la complejidad.
- Facilita el mantenimiento y extensión de los sistemas.
- Es más adecuado para la construcción de entornos GUI.

Fomenta la reutilización, con gran impacto sobre la productividad y confiabilidad.

Objetos

Para comprenderlo veamos la siguiente imagen y luego intentemos responder: ¿Cuáles son los objetos que se pueden abstraer para ver televisión? ¿Cómo podrías describirlos?



En el problema planteado se pueden identificar tres elementos: la persona, el televisor y el control remoto. Cada elemento posee sus propias características y comportamientos. En la POO a estos elementos se los conoce bajo el nombre de OBJETOS, a las características o estados que identifican a cada objeto ATRIBUTOS y, a los comportamientos o acciones, MÉTODOS.

ELEMENTO	DESCRIPCIÓN
Persona	Tiene sus propios atributos: Apellido, Nombre, Altura, género, Color ojos, Cabello, etc. Y tiene un comportamiento: Ver , escuchar, hablar, correr, saltar, etc.
Control Remoto	Tiene sus propios atributos: Tamaño, color, tipo, batería, etc. Y tiene un comportamiento: Enviar señal, codificar señal, cambiar canal, aumentar volumen, ingresar a menú, prender TV, etc.
Televisor	Tiene sus propios atributos: pulgadas, tipo, número parlantes, marca , etc. Y tiene un comportamiento: Decodificar señal, prender, apagar, emitir señal, emitir audio, etc.

Del ejemplo anterior podemos inferir el concepto de objeto:

Un objeto es una entidad (tangible o intangible) que posee características propias (atributos) y acciones o comportamientos (métodos) que realiza por sí solo o interactuando con otros objetos.

Sin embargo, además de las características (atributos) y acciones (métodos)… ¿Qué otras características podrías mencionar? Observa la siguiente imagen para analizar..



De acuerdo a la imagen anterior podemos decir que un objeto, además, se identifica por un nombre o identificador único que lo diferencia de los demás (en este caso puede ser el nro de serie), un estado (encendido, apagado), un tipo (televisor), nos abstrae dejándonos acceder sólo a las funciones encendido, apagado, cambiar canal, etc. y, por supuesto tiene un tiempo de vida.

En base a lo expuesto anteriormente podemos expresar las características generales de los objetos en POO:

- Se identifican por un nombre o identificador único que lo diferencia de los demás.
- Poseen estados.
- Poseen un conjunto de métodos.
- Poseen un conjunto de atributos.
- Soportan el encapsulamiento (nos deja ver sólo lo necesario).
- Tienen un tiempo de vida.
- Son instancias de una clase (es de un tipo).

Para hacer eso, los lenguajes de programación orientados a objetos (como TypeScripts) usan descriptores (plantillas) de entidades conocidas como *clases*.

Clases

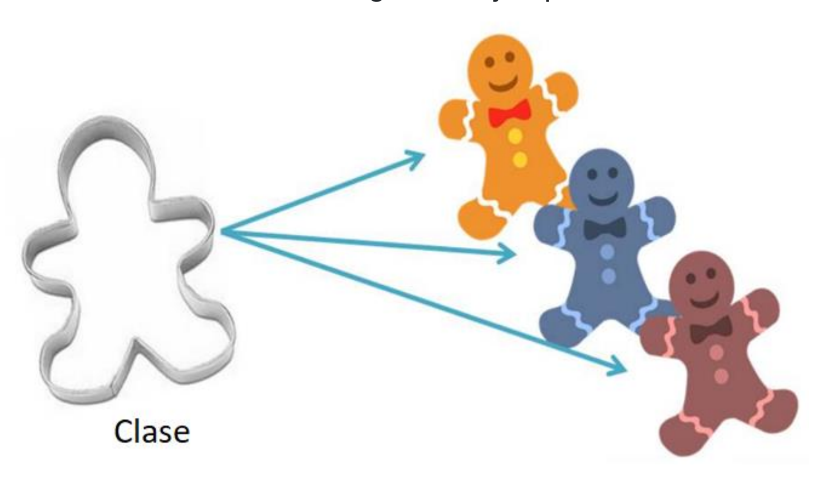
Una *clase* es la descripción de una entidad u objeto de forma tal que pueda usarse como plantilla para crear muchos objetos que respondan a dicha descripción. Para establecer analogías, se puede pensar que una clase se corresponde con el

concepto de *tipo de dato* de la programación estructurada tradicional, y los objetos creados a partir de la clase (llamados *instancias* en el mundo de la *POO*) se corresponden con el concepto de *variable* de la programación tradicional. Así como el *tipo* es uno solo y describe la forma que tienen todas las muchas *variables* de ese tipo, la *clase* es única y describe la forma y el comportamiento de los muchos *objetos* de esa clase.

Para describir objetos que responden a las mismas características de forma y comportamiento, se definen las *clases*.

Veamos el siguiente ejemplo:





Platzi. (s. f.). *Platzi: Cursos online profesionales de tecnología*. Recuperado 11 de octubre de 2021, de https://platzi.com/clases/1629-java-oop/21578-abstraccion-que-es-una-clase/

En el mismo podemos observar un molde para hacer galletitas (la clase) y el resultado que consiste en una o más galletas (objeto o instancia de clase).

Características generales de las clases en POO.

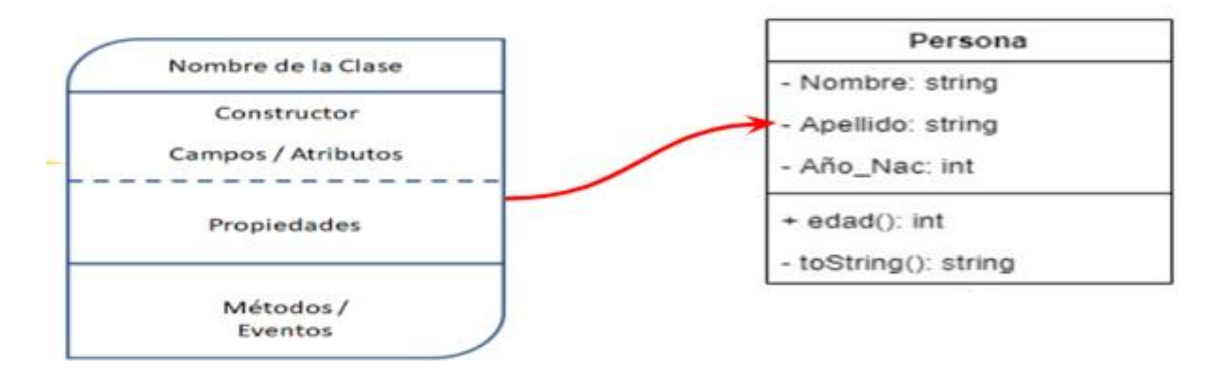
- Poseen un alto nivel de abstracción.
- Se relacionan entre sí mediante jerarquías.
- Los nombres de las clases deben estar en singular.

Representación gráfica de clases

La representación gráfica de una o varias clases se realiza mediante los denominados Diagramas de Clase. Para ello, se utiliza la notación que provee el Lenguaje de Modelación Unificado (UML, ver www.omg.org), a saber:

- Las clases se denotan como rectángulos divididos en tres partes. La primera contiene el nombre de la clase, la segunda contiene los atributos y la tercera los métodos.
- Los modificadores de acceso a datos y operaciones, a saber: público, protegido y privado; se representan con los símbolos +, # y –

respectivamente, al lado derecho del atributo. (+ público, # protegido, - privado).



Estructura de una clase (Diagrama de clases)

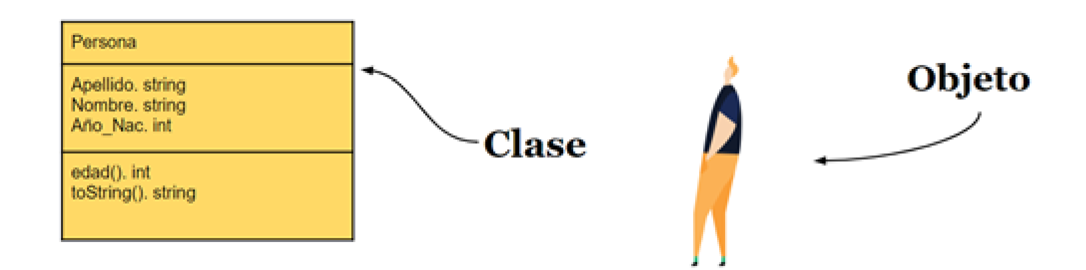
Relación entre clases y objetos

Algorítmicamente, las clases son descripciones netamente estáticas o plantillas que describen objetos. Su rol es definir nuevos tipos conformados por atributos y operaciones. Es decir que, las clases son una especie de molde de fábrica, en base al cual son construidos los objetos.

Por el contrario, los objetos son instancias particulares de una clase. Durante la ejecución de un programa sólo existen los objetos, no las clases.

- La declaración de una variable de una clase NO crea el objeto.

La creación de un objeto debe ser indicada explícitamente por el programador (instanciación), de forma análoga a como inicializamos las variables con un valor dado, sólo que para los objetos se hace a través de un método CONSTRUCTOR.



Relación entre clases y objetos.







Clases en Typescript

Clases

Una clase en Typescript no es más que una secuencia de símbolos (o caracteres) de un alfabeto básico. Esta secuencia de símbolos forma lo que se denomina el código fuente de la clase. Hay dos aspectos que determinan si una secuencia de símbolos es correcta en Typescript: la **sintaxis** y la **semántica**.

Las reglas de sintaxis de Typescript son las que permiten determinar de qué manera los símbolos del vocabulario pueden combinarse para escribir código fuente correcto mientras que la semántica por su parte guarda una estrecha relación con las acciones o instrucciones lo que permite determinar el significado de la secuencia de símbolos para que se lleve a cabo la acción por la computadora.

Así, por ejemplo, como en el lenguaje natural son las reglas de la sintaxis las que nos permiten determinar que la siguiente secuencia *"***robot el hombre programa***"* no es correcta; las reglas semánticas nos posibilitan detectar errores de interpretación que no permiten que las acciones o instrucciones puedan ser ejecutadas. Ejemplo: *"***el robot programa al hombre***"* es sintácticamente correcta pero **no semánticamente correcta**.

Sintaxis para la definición de clases en Typescript

```
class <nombre de la clase>
{
/* Atributos */
/* Métodos */
}
```

Dentro de las clases podemos encontrar:

Modificadores de Acceso

```
class Persona{
    readonly nombre:string;
    readonly apellido:string;
    private añoNac:number;
    constructor(nombre:string, apellido:string) {
        this.nombre = nombre;
        this.apellido = apellido;
      }
  public toString():string
  {
    return  this.nombre + this.apellido;
  }
  public edad(añoActual:number):number
  {
    return ( añoActual - this.añoNac);
  }
}
```

Atributos

Constructor

Métodos

Atributos: Son *variables* que se declaran dentro de la clase, y sirven para indicar la *forma o características* de cada objeto representado por esa clase. Los atributos, de alguna manera, muestran lo que cada objeto *es*, o también, lo que cada objeto *tiene*.

Sintaxis: ***<nombre_variable>: <tipo_de_datos>,*** *ejemplo:*

```
class Personal{
    //Atributos de la clase Persona
    nombre:string;
    apellido:string;
    añoNac:number;
  }
```

Métodos: Son funciones, procedimientos o rutinas declaradas dentro de la clase, usados para describir el *comportamiento o las acciones* de los objetos descriptos por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto *hace*.

La sintaxis es:

<nombre_método>(<parámetros>): <tipo_de_datos_devuelto>,

{

//**instrucciones*/

}

Ejemplo:

```
EdadAproximada(añoActual:number):number
{
  return añoActual - this.añoNac;
}
```

Dentro de estas <instrucciones> se puede acceder a todos los miembros definidos en la clase, a la cual pertenece el método. A su vez, todo método puede devolver un objeto como resultado de haber ejecutado las <instrucciones>. En tal caso, el tipo del objeto devuelto tiene que coincidir con el especificado en <TipoDevuelto>, y el método tiene que terminar con la cláusula return <objeto>;

Para el caso que se desee definir un método que no devuelva objeto alguno se omite la cláusula return y se especifica void como <TipoDevuelto>.

Opcionalmente todo método puede recibir en cada llamada una lista de parámetros a los que podrá acceder en la ejecución de las <instrucciones> del mismo. En <Parámetros> se indican los tipos y nombres de estos parámetros y es mediante estos nombres con los que se deberá referirse a ellos en el cuerpo del método.

Es común (pero no obligatorio) que los atributos de la clase se declaren antes que los métodos. El conjunto de atributos y métodos de una clase se conoce como el conjunto de ***miembros*** de la clase.

Finalmente es importante mencionar que las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan relacionar entre sí, de manera que puedan compartir atributos y métodos sin necesidad de volver a escribirlos.

Constructores: es un método especial que permite instanciar un objeto. Su nombre está definido por la palabra **constructor**, y no tiene ningún tipo de retorno. Puede recibir 0 a n parámetros.

Sintaxis:

Constructor((<parámetros>: <tipo_de_datos_devuelto>))

{

/**instrucciones*/

}

Ejemplo:

```
constructor(nombre:string, apellido:string) {
    this.nombre = nombre;
    this.apellido = apellido;
  }
```

Este código suele usarse para la inicialización de los atributos del objeto a crear, sobre todo cuando el valor de éstos no es constante o incluye acciones más allá de una asignación de valor.

Propiedades (getters y los setters). Las mismas proporcionan la comodidad de los miembros de datos públicos sin los riesgos que provienen del acceso sin comprobar, sin controlar y sin proteger a los datos del objeto.

Ejemplo:

```
get Nombre():string {
    return this.nombre;
}

set Nombre (nombre:string){
    this.nombre=nombre;
}

get Apellido():string {
    return this.apellido;
}

set Apellido(apellido:string){
    this.apellido=apellido;
}

get AñoNacimiento():number {
    return this.añoNac;
}

set AñoNacimiento(añoNac:number){
    this.añoNac=añoNac;
}
```

Modificadores de acceso: La forma que los programas orientados a objetos, provee para que un programador obligue a respetar el Principio de Ocultamiento son los llamados *modificadores de acceso*.

Se trata de ciertas palabras reservadas que colocadas delante de la declaración de un atributo o de un método de una clase, hacen que ese atributo o ese método tengan accesibilidad más amplia o menos amplia desde algún método que no esté en la clase. Así, los modificadores de acceso pueden ser: *public*, *private* , *protected*

- **Public**: un miembro público es accesible tanto desde el interior de la clase (por sus propios métodos), como desde el exterior de la misma (por métodos de otras clases).
- **Private**: sólo es accesible desde el interior de la propia clase, usando sus propios métodos.
- **Readonly**: El acceso es de sólo lectura.
- **Protected**: aplicable en contextos de herencia (tema que veremos más adelante), hace que un miembro sea público para sus clases derivadas y para clases en el mismo paquete, pero los hace privados para el resto.

Sintaxis: ***<modificador> <atributo o método>,*** ejemplo:

```
class Personal{
    //Atributos de la clase Persona
    private nombre:string;
    private apellido:string;
    private añoNac:number;
  }
```

Decoradores e instancias en TypeScript

Decoradores de Clase

En Typescript, los decoradores (decorators en inglés) permiten añadir anotaciones y metadatos o incluso cambiar el comportamiento de clases, propiedades, métodos y parámetros.

Un decorador no es más que **una función** que, dependiendo de qué cosa queramos decorar, sus argumentos serán diferentes.
Ejemplo:

```
function DecoradorPersona(target:Function) {
    console.log(target);
  }

  @DecoradorPersona
  class Persona{
    constructor() {
    ...
      }
  }
```

En el ejemplo imprimimos por consola la clase Persona que fue decorada.

Si necesitamos algo más avanzado, debemos pasar parámetros a los decoradores:

```
function DecoradorPersona(data:string) {
    return function <T extends { new(...args: any[]): {} }>(constructor: T) {
      return class extends constructor {

      array = data.split(",");
      Nombre=this.array[0];
      Apellido=this.array[1];
      }
    }
  }

  @DecoradorPersona('Juan,López')
  class Persona{
      private nombre:string="";
      private apellido:string="";
      private añoNac:number=0;

     constructor(nombre:string, apellido:string) {
      this.nombre = nombre;
      this.apellido=apellido;
     }
  ...
```

Al momento de instanciar el objeto a través de su constructor y luego acceder a la propiedad Nombre, podremos observar que ésta fue reemplazada por Juan López.

```
    let persona=new Persona("Juan,
López");
    console.log(persona.toString());
```

Ejemplo de decoración que cambia el valor de la variable nombre.

<u>**Nota**</u>: Es posible también decorar métodos, propiedades, etc.

Instancias

Para manipular los objetos o instancias de las clases (tipos) también se utilizan variables y éstas tienen una semántica propia la cual, se diferencia de los tipos básicos. Para ello, deberemos usar explícitamente el operador NEW. En caso contrario contendrán una referencia a null, lo que semánticamente significa que no está haciendo referencia a ningún objeto.

Sintaxis para instanciar objetos: ***<nombre_objeto>= new <Nombre_de_Clase>(<parámetros>),*** ejemplo:








```
let persona= new Persona();
```

Sintaxis para inicializar un objeto:

Hay 3 maneras de inicializar un objeto. Es decir, proporcionar datos a un objeto.

1. Por referencia a variables, ejemplo:

```
let persona= new Persona();
persona.apellido="Rosas";
persona.nombre ="Maria";
```

2. Por medio del constructor de la clase, ejemplo:

```
let persona= new Persona("Maria","Rosas");
```

3. Por medio de la propiedad setter, ejemplo:

```
let persona= new Persona();
persona.Apellido="Rosas";
persona.Nombre ="Maria";
```

Recomendaciones

Aunque cada programador puede definir su propio estilo de programación, una buena práctica es seguir el estilo utilizado por los diseñadores del lenguaje pues, de seguir esta práctica será mucho más fácil analizar el fuente de terceros y, a su vez, que otros programadores analicen y comprendan nuestro fuente.

- Evitar en lo posible líneas de longitud superior a 80 caracteres.
- Indentar los bloques de código.
- Utilizar identificadores nemotécnicos, es decir, utilizar nombres simbólicos adecuados para los identificadores lo suficientemente autoexplicativos por sí mismos para dar una orientación de su uso o funcionalidad de manera tal que podamos hacer más claros y legibles nuestros códigos.
- Los identificadores de clases, módulos, interfaces y enumeraciones deberán usar **PascalCase**.
- Los identificadores de objetos, métodos, instancias, constantes y propiedades de los objetos deberán usar camelCase.

- Utilizar comentarios, pero éstos seguirán un formato general de fácil portabilidad y que no incluya líneas completas de caracteres repetidos. Los que se coloquen dentro de bloques de código deben aparecer en una línea independiente indentada de igual forma que el bloque de código que describen.

Fundamentos del enfoque orientado a objetos (EOO)

El Enfoque Orientado a Objeto se basa en cuatro principios que constituyen la base de todo desarrollo orientado a objetos. Estos principios son:

- Jerarquías (herencia, agregación y composición)
- Abstracción
- Encapsulamiento
- Modularidad

Otros elementos para destacar (aunque no fundamentales) son:

- Polimorfismo
- Tipificación
- Concurrencia
- Persistencia.

Jerarquías

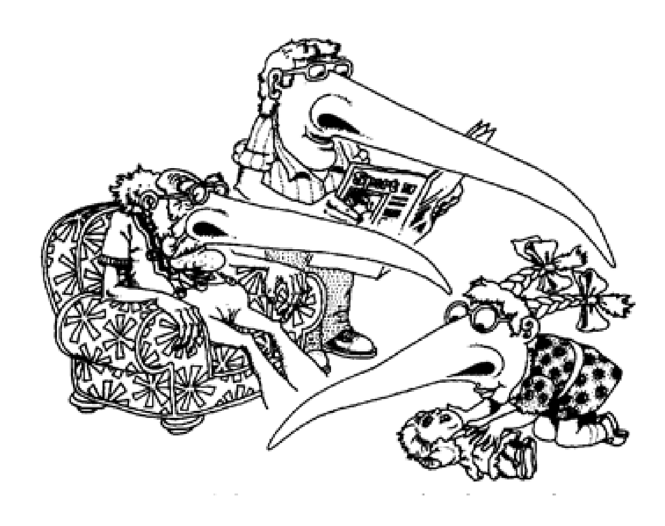
Las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan relacionar entre sí de manera que puedan compartir atributos y métodos sin necesidad de volver a escribirlos y así resolver un problema.

La posibilidad de establecer jerarquías entre las clases es una característica que diferencia esencialmente la programación orientada a objetos de la programación tradicional, ello debido fundamentalmente a que permite extender y reutilizar el código existente sin tener que volver a escribirlo cada vez que se necesite.

Herencia

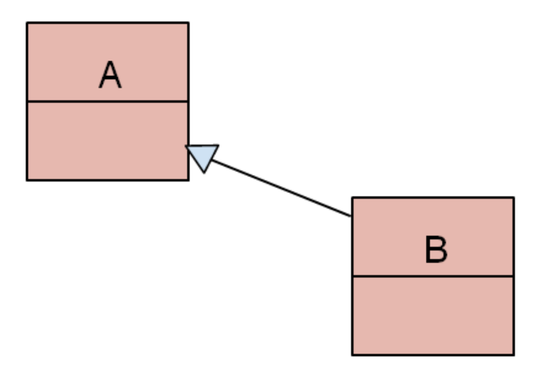
En Programación Orientada a Objetos se llama *herencia* al mecanismo por el cual se puede definir una nueva clase *B* en términos de otra clase *A* ya definida, pero de forma que la clase *B* obtiene todos los miembros definidos en la clase *A* sin necesidad de hacer una redeclaración explícita. El sólo hecho de indicar que la clase *B* hereda (o deriva) desde la clase *A*, hace que la clase *B* incluya todos los miembros de *A* como propios (a los cuales podrá acceder en mayor o menor medida de acuerdo al calificador de acceso [*public*, *private*, *protected*, *"default"*] que esos miembros tengan en *A*).

Es decir que la herencia permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos (Programación Diferencial), evitando repetición de código y permitiendo la reusabilidad.



Laura Álvarez, Helmer Avendaño, Yeison García, Sebastián Morales, Edwin Bohórquez, Santiago Hernandez, Sebastián Moreno, Cristian Orjuela (2017) https://ferestrepoca.github.io/paradigmas-de-programacion/poo/poo_teoria/concepts.html

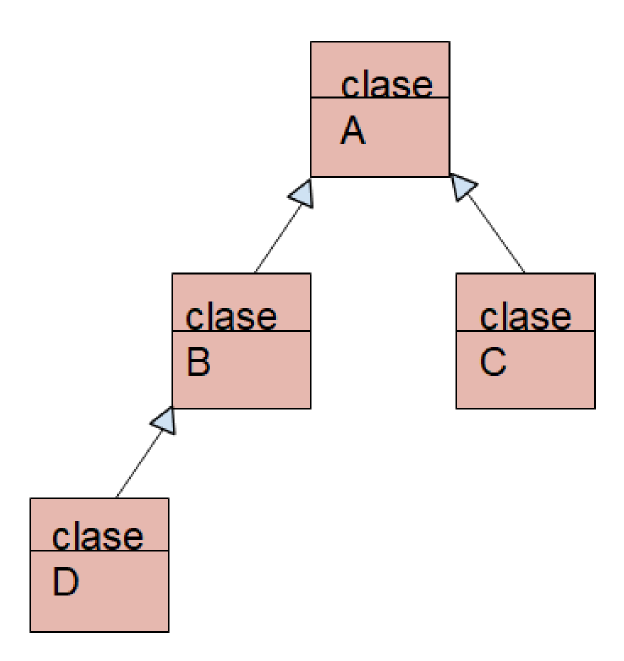
Cuando la clase *B* hereda de la clase *A*, se dice que hay una *relación de herencia* entre ellas, y se modela en UML con una flecha continua terminada en punta cerrada. La flecha parte de la nueva clase (o clase derivada) que sería *B* en nuestro ejemplo, y termina en la clase desde la cual se hereda (que es *A* en nuestro caso):



La clase desde la cual se hereda se llama ***super clase***, y las clases que heredan desde otra se llaman *subclases* o *clases derivadas*: de hecho, la herencia también se conoce como *derivación de clases*.

Una ***jerarquía*** *de clases* es un conjunto de clases relacionadas por herencia. La clase en la cual nace la jerarquía que se está analizando se designa en general como *clase base* de la jerarquía. La idea es que la *clase base* reúne en ella características que son comunes a todas las clases de la jerarquía, y que por lo

tanto todas ellas deberían compartir sin necesidad de hacer redeclaraciones de esas características. El siguiente gráfico muestra una jerarquía de clases:



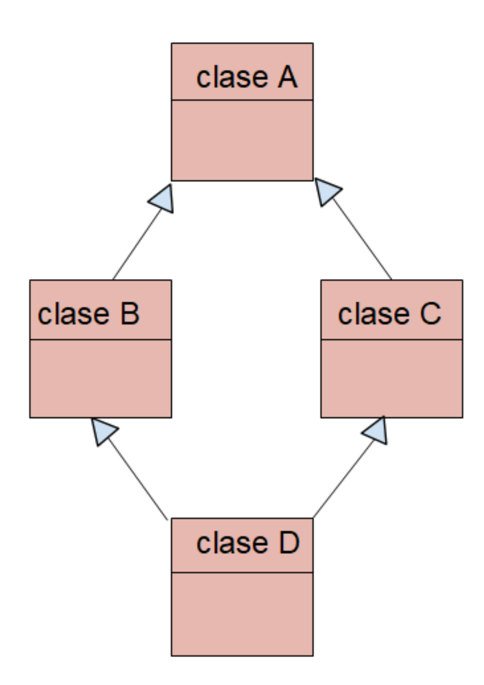
En esta jerarquía, la *clase base* es "*Clase A*". Las clases "*Clase B*" y "*Clase C*" son *derivadas directas* de "*Clase A*". Note que "*Clase D*" deriva en forma directa desde "*Clase B*", pero en *forma indirecta* también deriva desde "*Clase A*", por lo tanto todos los elementos definidos en "*Clase A*" también estarán contenidos en "*Clase D*". Siguiendo con el ejemplo, "*Clase B*" es super clase de "*Clase D*", y "*Clase A*" es super clase de "*Clase B*" y "*Clase C*".

En general, se podrían definir esquemas de jerarquías de clases en base a dos categorías o formas de herencia:

Herencia simple: Si se siguen reglas de *herencia simple*, entonces *una clase puede tener una y sólo una superclase directa*. El gráfico anterior es un ejemplo de una jerarquía de clases que siguen *herencia simple*. La *Clase D* tiene una sola superclase directa que es *Clase B*. No hay problema en que a su vez esta última derive a su vez desde otra clase, como *Clase A* en este caso. El hecho es que, en *herencia simple*, a nivel de gráfico UML, sólo puede existir *una* flecha que parta desde la clase derivada hacia alguna superclase.

Herencia múltiple: Si se siguen reglas de *herencia múltiple*, entonces *una clase puede tener tantas superclases directas como se desee*. En la gráfica UML, puede haber varias flechas partiendo desde la clase derivada hacia sus superclases. El siguiente esquema muestra una jerarquía en la que hay *herencia múltiple*: note que *Clase D* deriva en forma directa desde las clases *Clase B* y *Clase C*, y esa situación es un caso de herencia múltiple. Sin embargo, note también que la relación que existe entre las *Clase B* y *Clase C* contra *Clase A* es de *herencia*

simple; tanto *Clase B* como *Clase C* tienen una y sólo una superclase directa: *Clase A*.

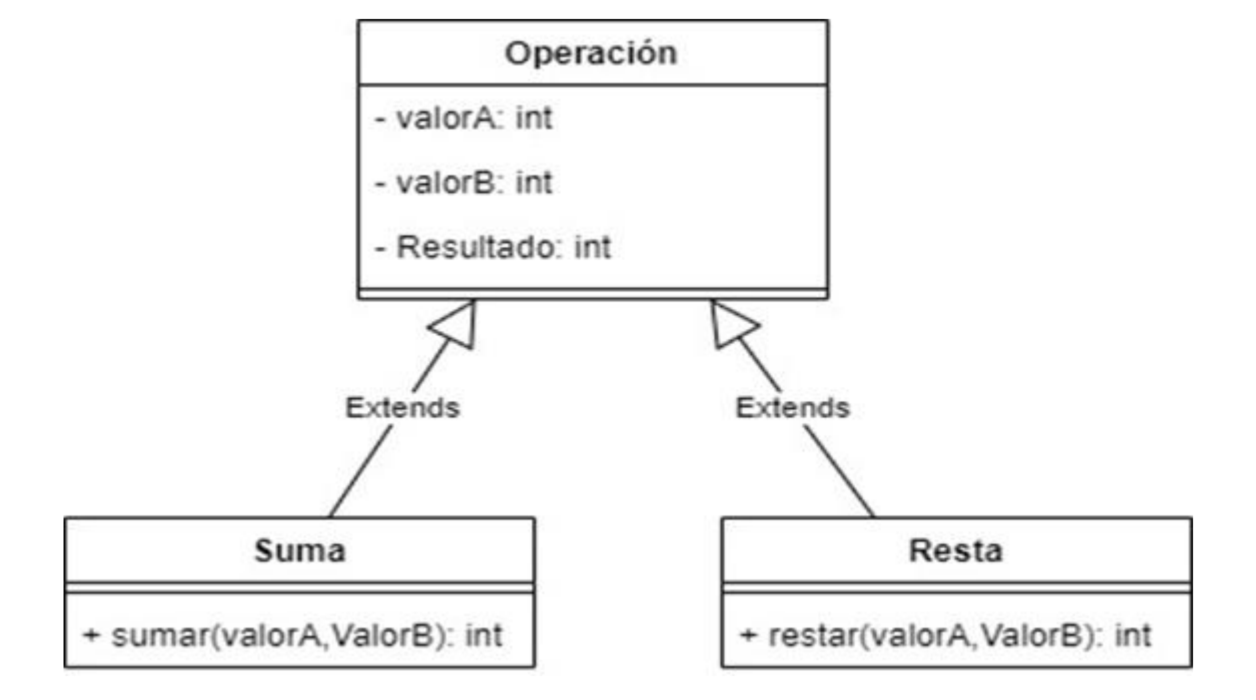


No todos los lenguajes orientados a objetos soportan (o permiten) la herencia múltiple: este mecanismo es difícil de controlar a nivel de lenguaje y en general se acepta que la herencia múltiple lleva a diseños más complejos que los que se podrían obtener usando sólo herencia simple y algunos recursos adicionales como la implementación de *clases de interface.*

Sabemos que una *relación de uso* implica que un objeto de una clase *usa* a un objeto de otra. Pero una *relación de herencia* implica que un objeto *b* de una clase *B*, *es* a su vez un objeto de otra clase *A*.

Veamos un ejemplo en Typescript:

Necesitamos crear dos clases que llamaremos Suma y Resta que derivan de una superclase llamada Operación:



En typescript, seguir los siguientes pasos:

1- Definir la superclase operación:

```
class Operacion{
    protected valorA:number;
    protected valorB:number;
    protected resultado:number;
    constructor() {
        this.valorA=0;
        this.valorB=0;
        this.resultado=0;
      }

    set ValorA(value:number){
        this.valorA=value;
    }
    set ValorB(value:number){
        this.valorB=value
    }

    get Resultado():number {
    return this.resultado;
  }
}
```

2- Luego, extender las subclases suma y resta:







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein

```
class Suma extends Operacion
{
    Sumar ()
    {
        this.resultado=this.valorA+this.valorB;
    }
}
```

```
class Restar extends Operacion
{
    Restar ()
    {
        this.resultado=this.valorA-this.valorB;
    }
}
```

Observa que debemos utilizar la palabra “***extends***”

3- Crear instancias de la clase suma y resta:

```
let operacionS= new Suma();
operacionS.ValorA=45;
operacionS.ValorB=10;
operacionS.Sumar();
console.log("El resultado de la suma es " + operacionS.Resultado);
```

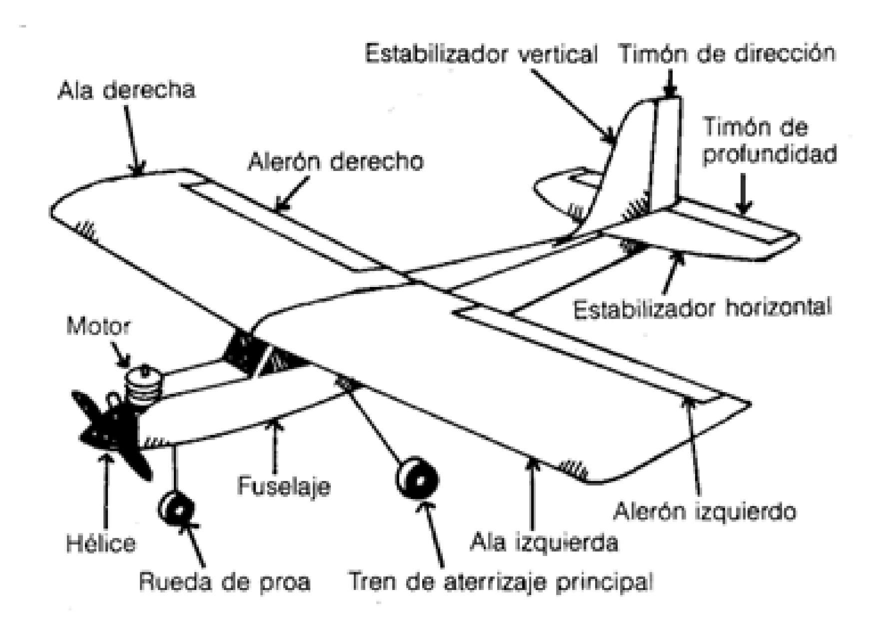
```
let operacionR= new Resta();
operacionR.ValorA=45;
operacionR.ValorB=10;
operacionR.Restar();
console.log("El resultado de la resta es " + operacionR.Resultado);
```

Nota: observa que los modificadores de acceso en la superclase son ***“protected”***. Estos permiten que la subclase pueda acceder a ellos y manipularlos.

Agregación y Composición

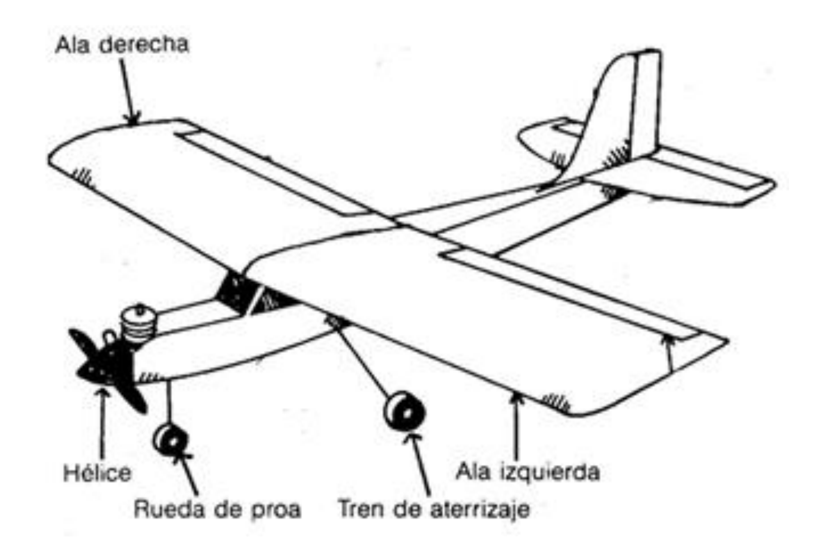
Las jerarquías de agregación y composición son asociaciones entre clases del tipo “es parte de”.

Para comprenderlo observemos la siguiente imagen de un aeroplano:



Fuente de la imagen: https://hobbymodels.com.mx/blog/aviones/

El mismo está compuesto de partes que pueden ser elementales (composición) o no (agregación).



En la imagen anterior podemos observar que el aeroplano está compuesto por:

- 1 hélice frontal.
- Tren de aterrizaje fijo, tiene 3 neumáticos y 3 amortiguadores.

- 2 alas frontales y 3 de cola.
- La cubierta cuenta con sólo una cabina de vuelo, 1 tanque de combustible, 1 puerta de salida.

Para resolverlo en Typescript, deberemos primero crear las clases Helice, TrenDeAterrizaje, Turbina, Cubierta, Alas, etc.

A continuación, ejemplo de las clases Turbina y Cubierta en Typescript

```
class Turbina
{
    private  numTurbinas:number = 0;
    public constructor( n :number)
    {
        this.numTurbinas = n;
    }
    public  ToString()
    {
        return this.numTurbinas + " Turbina/s";
    }
}
```

```
class Cubierta

 {

    private cabinaTripulacion:boolean = false;

    private cabinaVuelo:boolean = false;

    private sistemaEmergencia:boolean = false;

    private numTanquesCombustible:number = 0;

    private numPuertasSalidas:number = 0;

    public constructor( pCabinaTripulacion:boolean,  pCabinaVuelo:boolean,  pSist
emaEmergencia:boolean,  pTanquesCombustible:number,  pPuertasSalida:number
)

    {

       this.cabinaTripulacion = pCabinaTripulacion;

       this.cabinaVuelo = pCabinaVuelo;
```








```
        this.sistemaEmergencia = pSistemaEmergencia;
        this.numTanquesCombustible = pTanquesCombustible;
        this.numPuertasSalidas = pPuertasSalida;
    }

    public ToString()
    {
        let mensaje = "Cubierta compuesta de: ";
        if (this.cabinaVuelo)
        {
            mensaje += " Cubierta de Vuelo, ";
        }
        if (this.cabinaTripulacion)
        {
            mensaje += " Cubierta de Tripulación, ";
        }
        if (this.sistemaEmergencia)
        {
            mensaje += " Sistema de Emergencia, ";
        }
        mensaje += this.numTanquesCombustible + " Tanques de Combustible, ";
        mensaje += this.numPuertasSalidas + " Puertas de Salida.";
        return mensaje;
```

```
    }
  }
```

Las mismas forman parte elemental del Aeroplano (asociación de composición). Esta última se muestra a continuación:

```
class Aeroplano

{

   private  helice: Helice ;

   private  trenAterrizaje:TrenDeAterrizaje;

   private  alas: Alas ;

   private  cubierta:Cubierta ;



   constructor( phelice:Helice,  pTrenAterrizaje:TrenDeAterrizaje,  pAlas:Alas,  pCu
bierta:Cubierta)

   {

       this.helice = phelice;

       this.trenAterrizaje = pTrenAterrizaje;

       this.alas = pAlas;

       this.cubierta = pCubierta;

   }

   public  ToString()

   {

      let mensaje = "Aeroplano compuesto por: ";

      mensaje += this.helice.ToString();
```

```
        mensaje += this.alas.ToString();

        mensaje += this.trenAterrizaje.ToString();

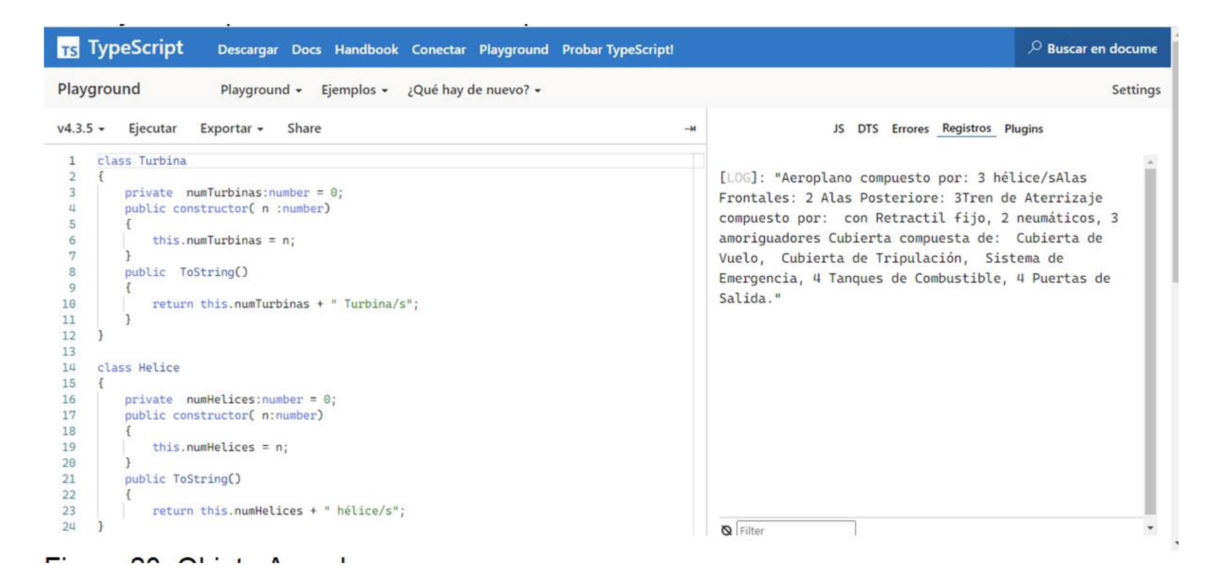
        mensaje += this.cubierta.ToString();

        return mensaje;


    }

}
```

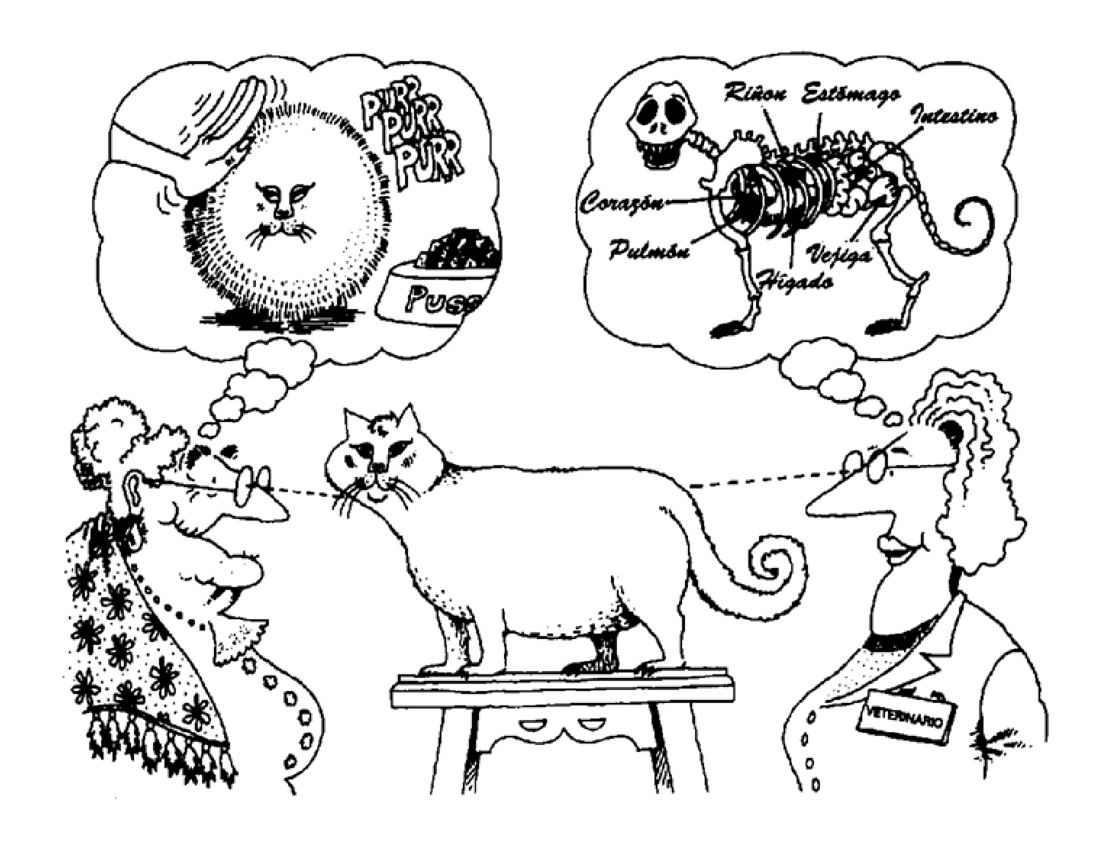
Si lo ejecutas el código podrás observar su composición:



Abstracción

Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo distingue de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

“Una abstracción se centra en la visión externa de un objeto por lo tanto sirve para separar el comportamiento esencial de un objeto de su implementación. La decisión sobre el conjunto adecuado de abstracciones para determinado dominio es el problema central del diseño orientado a objetos. Se puede caracterizar el comportamiento de un objeto de acuerdo a los servicios que presta a otros objetos, así como las operaciones que puede realizar sobre otros objetos”



Laura Álvarez, Helmer Avendaño, Yeison García, Sebastián Morales, Edwin Bohórquez, Santiago Hernandez, Sebastián Moreno, Cristian Orjuela (2017)

Los mecanismos de abstracción usados en el EOO para extraer y definir las abstracciones son:

1- La **GENERALIZACIÓN**. Mecanismo de abstracción mediante el cual un conjunto de clases de objetos son agrupados en una clase de nivel superior (Superclase), donde las semejanzas de las clases constituyentes (Subclases) son enfatizadas, y las diferencias entre ellas son ignoradas.

En consecuencia, a través de la generalización:

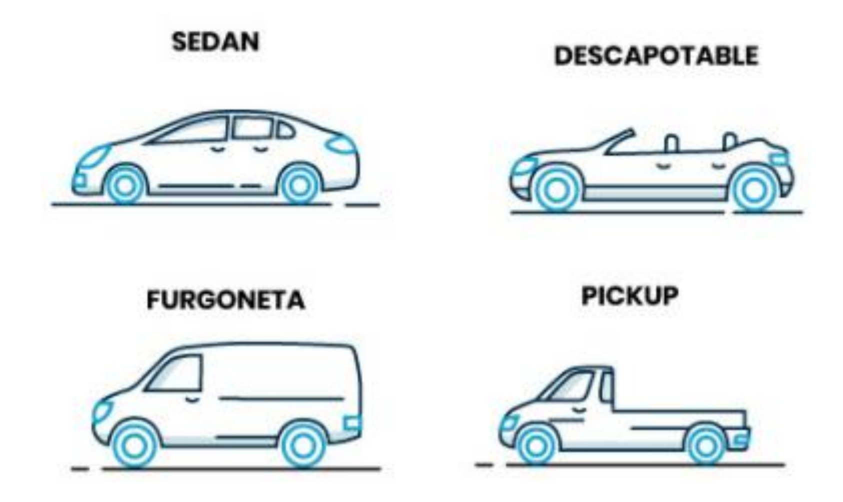
- La superclase almacena datos generales de las subclases
- Las subclases almacenan sólo datos particulares.

2- La **ESPECIALIZACIÓN** es lo contrario de la generalización. La clase Médico es una especialización de la clase Persona, y a su vez, la clase Pediatra es una especialización de la superclase Médico.

3- La **AGREGACIÓN**. Mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). Mediante agregación se puede definir por ejemplo un computador, por descomponerse en: la CPU, la ULA, la memoria y los dispositivos periféricos. El contrario de agregación es la descomposición.

4- La **CLASIFICACIÓN**. Consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La ejemplificación es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular

La clasificación es el medio por el que ordenamos, el conocimiento ubicado en las abstracciones, miremos algunos ejemplos en las siguientes imágenes.



En resumen, las clases y objetos deberían estar al nivel de abstracción adecuado. Ni demasiado alto ni demasiado bajo.

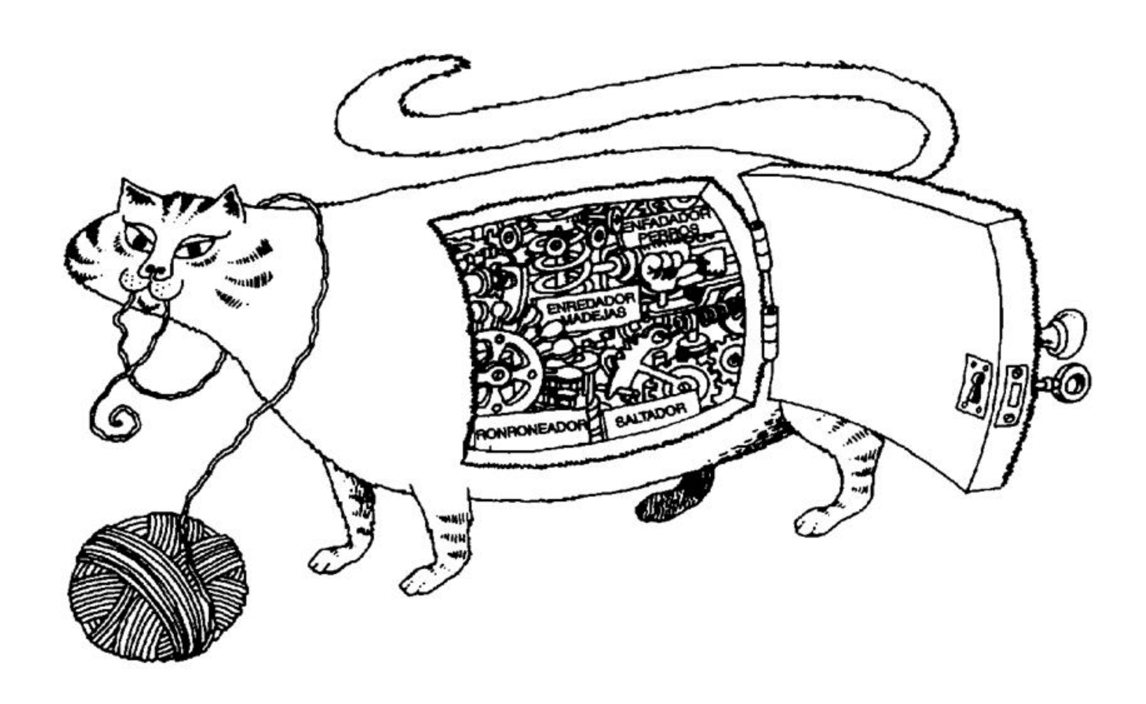
Vegetal
Equisetos
Licopodios
Gimnospermas
Angiospermas
Helechos
Musgos

Animal
Poríferos
Cnidarios
Platelmintos
Moluscos
Anélidos
Equinodermos
Insectos
Crustáceos
Arácnidos
Peces
Anfibios
Aves
Reptiles
Mamíferos

Encapsulamiento

También conocido como, Ocultamiento. Es la propiedad de la POO que permite ocultar los detalles de implementación del objeto mostrando sólo lo relevante. Esta parte de código oculta pertenece a la parte privada de la clase y no puede ser accedida desde ningún otro lugar.

El *encapsulamiento* da lugar al ya citado *Principio de Ocultamiento*: sólo los métodos de una clase deberían tener acceso directo a los atributos de esa clase, para impedir que un atributo sea modificado en forma insegura, o no controlada por la propia clase. El *Principio de Ocultamiento* es la causa por la cual en general los atributos se declaran como privados (*private*), y los métodos se definen públicos (*public*). Los calificadores *private* y *public* (así como *protected*, que se verá más adelante) tienen efecto a nivel de compilación: si un atributo de una clase es privado, y se intenta acceder a él desde un método de otra clase, se producirá en error de compilación.



Laura Álvarez, Helmer Avendaño, Yeison García, Sebastián Morales, Edwin Bohórquez, Santiago Hernandez, Sebastián Moreno, Cristian Orjuela (2017)

Ejemplo:

```
class Persona{
    private nombre:string;
    private apellido:string;
    private añoNac:number;
    constructor(nombre:string, apellido:string)
{
        this.nombre = nombre;
        this.apellido = apellido;
      }

  get Nombre():string {
    return this.nombre;
  }

  get Apellido():string {
    return this.apellido;
  }
  ...
}
```

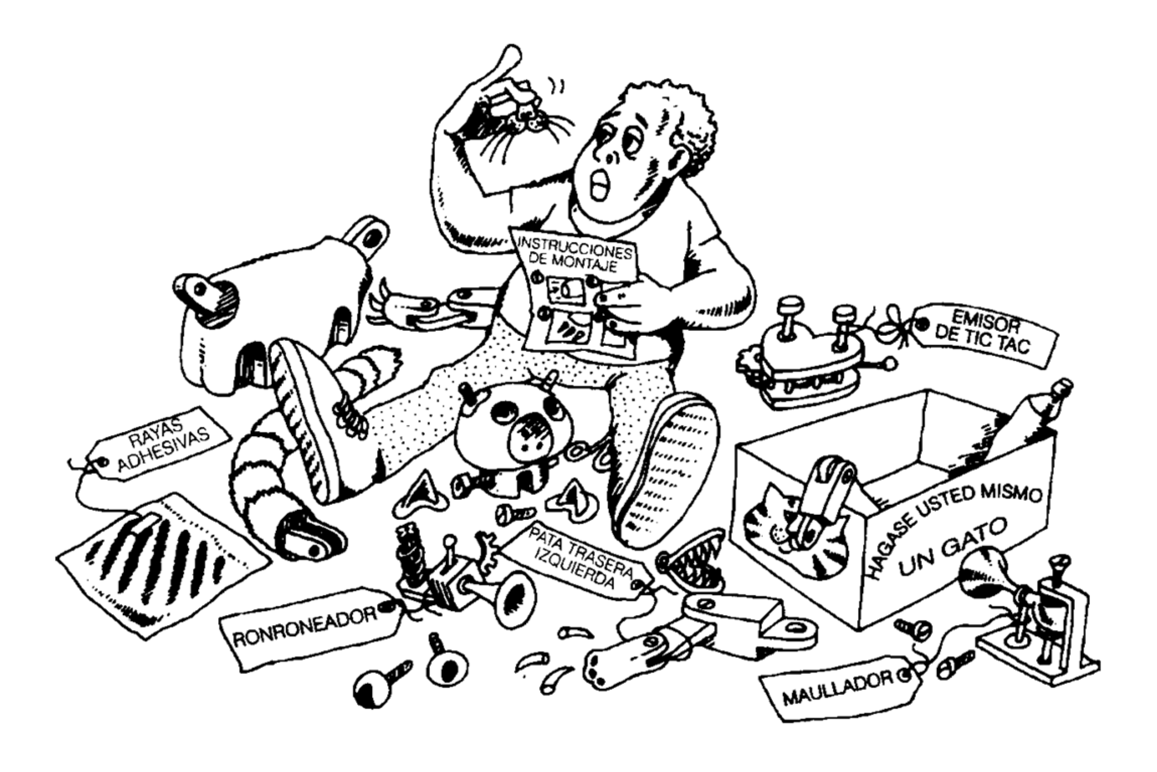
La clase Persona encapsula los atributos a fin de que, no tomen valores inconsistentes.

El encapsulamiento da lugar al Principio de Ocultamiento: sólo los métodos de una clase deberían tener acceso directo a los atributos de esa clase, para impedir que un atributo sea modificado en forma insegura, o no controlada por la propia clase.

Modularidad

Es la propiedad que permite tener independencia entre las diferentes partes de un sistema. La modularidad consiste en dividir un programa en módulos o partes, que pueden ser compilados separadamente, pero que tienen conexiones con otros

módulos. En un mismo módulo se suele colocar clases y objetos que guarden una estrecha relación. El sentido de modularidad está muy relacionado con el ocultamiento de información.



Laura Álvarez, Helmer Avendaño, Yeison García, Sebastián Morales, Edwin Bohórquez, Santiago Hernandez, Sebastián Moreno, Cristian Orjuela (2017)

Polimorfismo

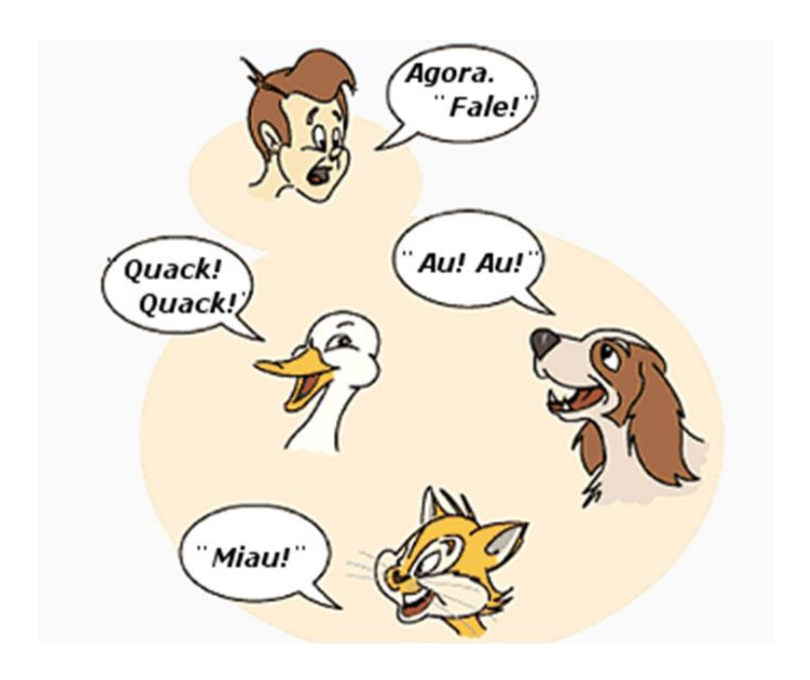
Clases diferentes (polimórficas) implementan métodos con el mismo nombre. En resumen, el polimorfismo permite comportamientos diferentes, asociados a objetos distintos compartiendo el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento







2021 Año de homenaje
al premio Nobel de medicina
Dr. César Milstein



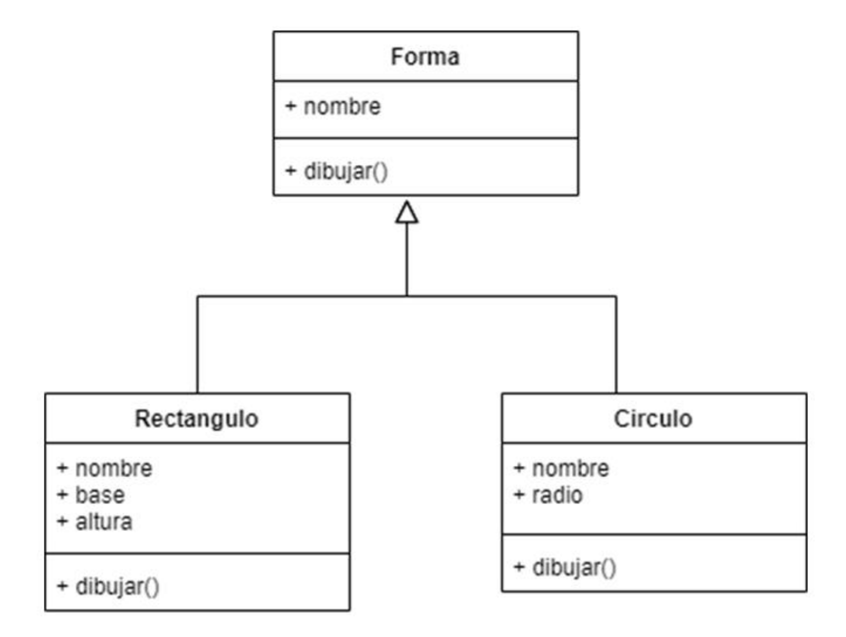
En el dibujo anterior puede verse la esencia del polimorfismo dado que todos implementan el método hablar aunque son diferentes objetos.

Existen diferentes formas de implementar el polimorfismo en typescript:

Polimorfismo por herencia

Cuando una subclase hereda de una clase base, obtiene todos los métodos, campos, propiedades y eventos de la superclase sin embargo, quizás necesitemos un comportamiento diferente para las clases derivadas (o subclases).

Ejemplo:









Del diagrama de clases anterior podemos deducir que no será lo mismo dibujar un rectángulo que un círculo por lo que el comportamiento deberá ser distinto (polimorfismo).

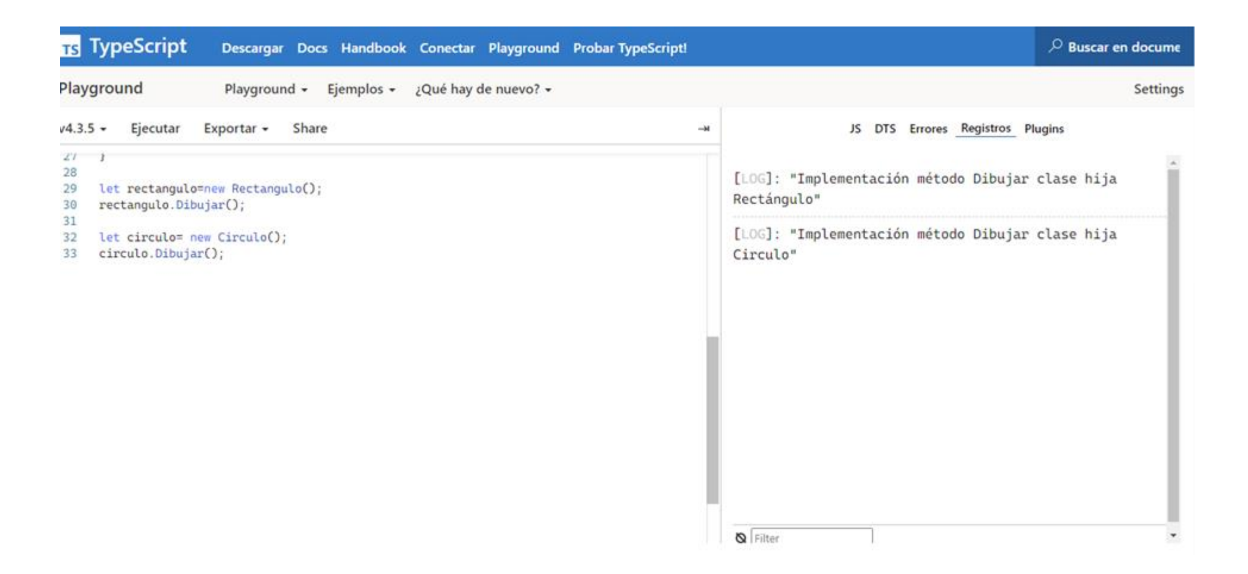
Ejemplo de polimorfismo en base a herencia en Typescript:

```
class Forma{
    nombre:string="";
    Dibujar()
    {
        console.log("Implementación método Dibujar clase base");
    }
}

class Rectangulo extends Forma
{
    base:number=0;
    altura:number=0;
    Dibujar()
    {
        console.log("Implementación método Dibujar clase hija Rectángulo");
    }
}

class Circulo extends Forma
{
    radio:number=0;
    Dibujar()
    {
        console.log("Implementación método Dibujar clase hija Circulo")
    }
}
```

Si creamos una instancia de la clase Rectangulo y Circulo y luego llamamos el método Dibujar observaremos que el mismo fue reemplazado por la implementación correspondiente para la forma:

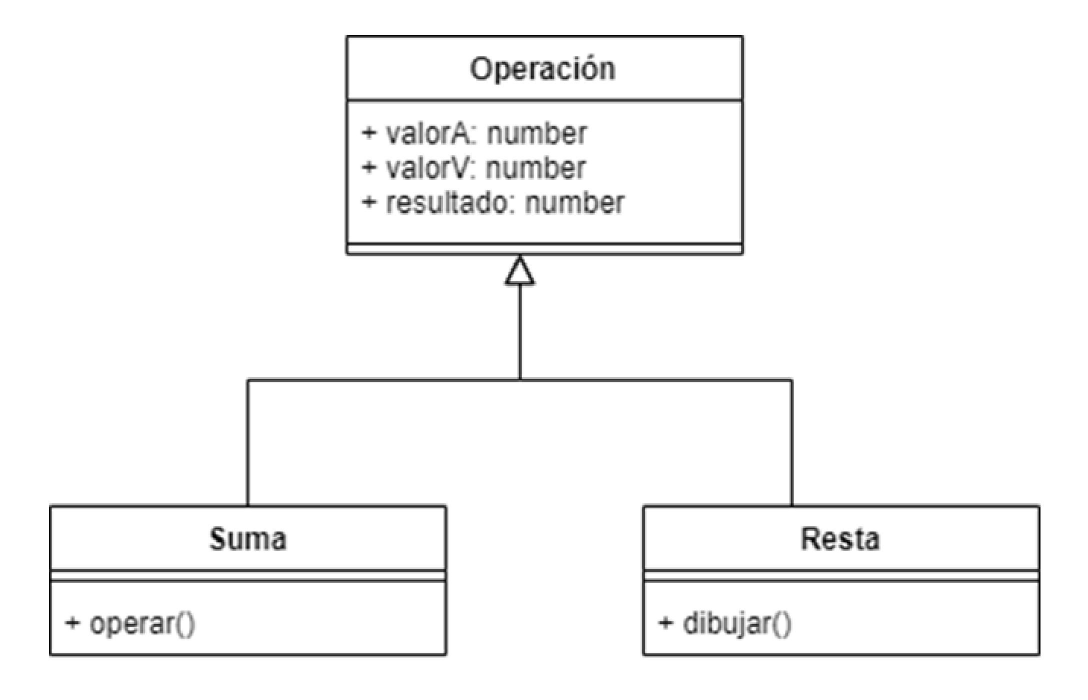


Polimorfismo por abstracción

El polimorfismo por abstracción consiste en definir clases base abstractas (que no se pueden instanciar) pero que sirven de base para las clases derivadas. Es decir, sólo existen para ser heredadas.

<u>**Nota**</u>: Las clases abstractas no están implementadas o si lo están es parcialmente. Forzosamente se ha de derivar si se desea crear objetos de la misma ya que_no es posible crear instancias a partir de ellas.

Veamos el siguiente diagrama de clases ejemplo:



En el mismo podemos observar una clase abstracta llamada Operación la cual define 3 atributos. Luego, tenemos otras dos clases que heredan de esta: Suma y Resta. Ambas implementan el método Operar. Sin embargo, el comportamiento del método Operar deberá diferir si estamos hablando de sumas o restas.

```
abstract class Operacion{
    protected valorA:number;
    protected valorB:number;
    protected resultado:number;
    abstract Operar():void;

    set ValorA(value:number){
        this.valorA=value;
    }
    set ValorB(value:number){
        this.valorB=value
    }

    get Resultado():number {
    return this.resultado;
   }


}
```

Como podemos observar, debemos definir la clase abstracta anteponiendo el modificador "***abstract"*** previo a la definición de la clase y al método que deseamos que forzosamente en las clases derivadas sea implementado.

Ejemplo:

```
class Suma extends Operacion
{
    Operar ()
    {
        this.resultado= this.valorA + this.valorB;
    }
}
```

```
class Resta extends Operacion
{
    Operar ()
    {
        this.resultado= this.valorA - this.valorB;
    }
}
```

Polimorfismo por interfaces

Recordemos que una interfaz es un CONTRATO por lo que define propiedades y métodos, pero no su implementación.

Las interfaces, como las clases, definen un conjunto de propiedades, métodos y eventos. Pero de forma contraria a las clases, las interfaces no proporcionan implementación. Se implementan como clases y se definen como entidades separadas de las clases. Una interfaz representa un contrato, en el cual una clase que implementa una interfaz debe implementar cualquier aspecto de dicha interfaz exactamente como esté definido.

En typescript:

```
interface IOperacion{
    Operar(a:number,b:number):number;
}
```

Para implementar la interfaz en nuestra clase Suma y Resta (y lograr el polimorfismo) debemos utilizar la palabra ***implements*** como sigue:

```
class Suma implements IOperacion{
    Operar(a:number,b:number):number{
        return a+b;
    }
}

class Resta implements IOperacion{
    Operar(a:number,b:number):number{
        return a-b;
    }
}
```

Tipificación

Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas.

Concurrencia

La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.

La concurrencia permite a dos objetos actuar al mismo tiempo.

Persistencia

La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

Conserva el estado de un objeto en el tiempo y en el espacio.

Clases estáticas

Una static class es aquella clase que se usa sin necesidad de realizar una instanciación de la misma. Se utiliza como una unidad de organización para métodos no asociados a objetos particulares y separa datos y comportamientos que son independientes de cualquier identidad.

Desafortunadamente no existen clases estáticas en Typescript, aunque sí podemos definir sus métodos como estáticos y trabajar con ella sin instanciar el objeto.

```
class MyStaticClass {
    public static myMethod(): void { console.log("método estático")};
  }
```

Interfaces

Además de ser útiles para implementar el polimorfismo, las interfaces nos permiten crear nuevos tipos y de esta manera comprobar los tipos de las variables. Ej cuando se pasan como argumentos.

Ejemplo:

```
interface a {
    b: number;
    }
interface b extends a {
    c: string;
}
class test implements b {
    b: number;
    c: string;
    constructor (b: number, c: string) {
    this.b = b;
    this.c = c;
    }
}
```

En el ejemplo anterior podemos observar que la clase test hereda de b quien a su vez, hereda de a. Entonces, test tiene acceso

Typescripts permite crear interfaces según cuantas sean necesarias permitiéndonos evitar el anidamiento de objetos.

Ejemplo Interfaz Persona:

```
interface IPersona {
  nombre: string;
  edad: number;
  direccion:{
    calle: string;
    pais:string;
    ciudad:string;
  },
  mostrarDireccion:()=>string;
}
```

Instanciando un objeto en función de su interfaz tenemos:

```
const persona: IPersona ={
    nombre: 'Jose',
    edad:30,
    direccion:{
    calle: 'San Martin',
    pais:'Argentina',
    ciudad: 'Córdoba'
  },
  mostrarDireccion(){
    return this.nombre+','+this.direccion.ciudad+','+ this.direccion.pais;
  }
 console.log(persona.mostrarDireccion());
```

De lo anterior podemos observar que, tenemos una interfaz IPersona que anida la declaración de “direccion”.

TypeScript | Descargar | Docs | Handbook | Conectar | Playground | Probar TypeScript! | Buscar en docume

Playground | Playground | Ejemplos | ¿Qué hay de nuevo? | Settings

v4.3.5 | Ejecutar | Exportar | Share — JS | DTS | Errores | Registros | Plugins

```
    pais:string;
    ciudad:string;
  },
  mostrarDireccion:()=>string;
}

  const persona: Persona ={
    nombre: 'Jose',
    edad:30,
    direccion:{
    calle: 'San Martin',
    pais:'Argentina',
    ciudad: 'Córdoba'
  },
  mostrarDireccion(){
    return this.nombre+','+this.direccion.ciudad+','+ this.direccion.pais;
  }
  }

  console.log(persona.mostrarDireccion());
```

```
[LOG]: "Jose,Córdoba,Argentina"
```

Sin embargo, typescripts nos permite separar en más de una interfaz y así evitar el anidamiento que en algunos casos puede ser difícil de interpretar.

```
interface IPersona {
  nombre: string;
  edad: number;
  direccion: IDireccion,
  mostrarDireccion:()=>string;
}
interface IDireccion {
  calle: string;
  pais:string;
  ciudad:string;
}
  const persona: IPersona ={
    nombre: 'Jose',
    edad:30,
    direccion:{
    calle: 'San Martin',
    pais:'Argentina',
    ciudad: 'Córdoba'
  },
  mostrarDireccion(){
    return this.nombre+','+this.direccion.ciudad+','+ this.direccion.pais;
  }
  };
console.log(persona.mostrarDireccion());
```

Como podemos observar, la interfaz IPersona está compuesta por otra IDireccion evitando así un anidamiento difícil de seguir o comprender.